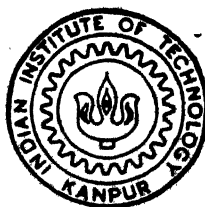


DESIGNING FOR PERFORMANCE IN RDBMS-BASED APPLICATIONS

By

G. C. PRASAD



Th
CSE/1993/M
P886d

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
December, 1993

DESIGNING FOR PERFORMANCE IN RDBMS-BASED APPLICATION

*A thesis submitted
in partial fulfillment of the requirements
for the degree of*

Master of Technology

by

G.C.Prasad

to the

Department of Computer Science and Engineering

Indian Institute of Technology, Kanpur

December, 1993

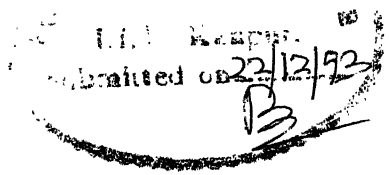
23 FEB 1994/CSE

CENTRAL BANK
1111 KANPUH

Doc. No. A. 117363

CSE-1983-M-PRA-DES

CERTIFICATE



It is certified that the work contained in the thesis titled *Designing for Performance in RDBMS-based Applications*, by G.C.Prasad, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

December, 1993

A handwritten signature in cursive script, reading 'T V Prabhakar'.

Dr. T.V.Prabhakar

Assistant Professor,

Department of Computer Science
and Engineering,

IIT Kanpur

ABSTRACT

Existing tools for database design do not adequately consider the effects of concurrency on performance, though contention for locks between concurrent transactions is one of the greatest causes of poor performance in RDBMS-based applications.

Most tools also focus exclusively on index selection. To solve performance problems, application implementors often have to make changes to the logical schema and to transactions, in addition to building indexes. Since these decisions are as important as index selection, if not more, design tools should ideally make recommendations in such areas as well. Currently, designers and administrators have to rely on their intuition and experience to make the decisions that are really crucial for performance.

This thesis attempts to develop a new design technique by focusing on lock conflict and the queueing of transactions that results from it. First, some typical concurrency-related problem patterns and their commonly-employed solutions are described, based on a study of some real-life RDBMS application sites. A diagrammatic tool is then introduced, which has the property of *visually* highlighting areas of lock conflict, and which thereby aids designers in choosing appropriate alternatives with the help of some heuristics. Data and process information are handled together in a single representation. Many design decisions that could not be arrived at using current design techniques follow logically using this method.

The viability of the approach has also been proven through a prototype computer-based design aid, which is briefly described.

Using this technique, even novice designers might be able to turn out more sophisticated designs.

ACKNOWLEDGEMENTS

The day I was selected to join the M.Tech. program at IIT-K, I met a few faculty members and discussed my areas of interest with them. When I asked about the possibility of doing my thesis on performance-oriented database design, one of them whole-heartedly and enthusiastically said yes, I could do it under his supervision.

That faculty member was Dr. T.V.Prabhakar, and our association in the year and a half since then has been extremely fruitful to me and enjoyable to, I trust, both of us.

With his eagerness to explore, youthful enthusiasm and endearing informality, he has encouraged me in many ways, even offering a new course just for the two students who showed interest in it. I thank him for the freedom and trust I received. I shall remember this always. I only regret we didn't have time to discuss birds and other wildlife, his knowledge of which, I am told, is as prodigious as his knowledge of databases.

I also thank my company CMC Ltd., not only for sponsoring me for the M.Tech. program, but for exposing me to the relational database environment which led in turn to all the rest. I hope this work can be used profitably by the project teams back home. Eshwaran and Gidds, if you ever read this, you'll know I'm not the same guy you could laugh at in 1988!

Heartfelt thanks to A.Dhingra of CMC Ltd., New Delhi, and to K.C.Bhushan, Ganesh Subramaniam, U.Phaniraj, Avinash Mane and Vinayak Pandit of CMC Ltd., Bombay, for their patient and excellent explanation of their projects. Most of the clever techniques detailed in this paper are due to them and their teams. Thanks also to all the senior managers at CMC Ltd., Bombay, for their cooperation and encouragement.

My belated thanks and apologies to Prof. S. Ankolekar (IIM, Ahmedabad). DBMS isn't a "DoBerMannS" after all, sir!

Sai and Narasimhan, thanks for patiently listening to Hindustani music with me, and not insisting on converting everything to CNF (Carnatic Normal Form). Nice knowing you, Mur'li (as you hate being called), my Classical Music foe and Science Fiction ally. And thanks for the company, mtech.*, you were a nice batch to be in.

Contents

1	Introduction	1
1.1	Traditional database design	1
1.2	The performance factor	1
1.3	A new attempt	4
2	Inadequacies of Earlier Approaches	6
2.1	Why concurrency and locking are important	6
2.2	Why the design of logical schema and transactions is more important than choosing an index configuration	7
2.3	Why transactions are more important than individual query statements . .	7
2.4	How vertical partitioning may sometimes aggravate performance problems .	8
2.5	Why a primary index need not be built on a primary key	9
3	Approach of This Thesis	11
3.1	Solving the real problem	11
3.2	Two ways to make a queue move faster	12
3.3	A design technique that neglects queueing - and pays for it	13
3.4	Clustering is better than you may think	14
3.5	Methodology of work	14
3.6	Contribution of this thesis	15
4	Case Studies	16
4.1	Schema (Logical and Physical) design decisions	16
4.1.1	Horizontal partitioning	16
4.1.2	Vertical partitioning	19

4.1.3	Redundancy	22
4.1.4	Clustering	24
4.1.5	Hashing/De-clustering	28
4.1.6	Indexes	29
4.2	Transaction and usage design decisions	30
4.2.1	Reading without locks	30
4.2.2	Splitting transactions	32
4.2.3	Pre-allocation of records	38
4.2.4	Archival	41
4.3	Miscellaneous techniques	43
5	Introducing a Diagrammatic Design Technique	44
5.1	Extensions to the standard entity-relationship diagram	44
5.2	Basic notation and methodology of usage	47
5.3	SQL shorthand	53
5.4	Modelling selection and conflict	53
6	Heuristics	55
7	Case Studies Revisited	60
7.1	Horizontal partitioning	60
7.2	Vertical partitioning	61
7.3	Redundancy	63
7.4	Clustering	69
7.5	Reading without locks	69
7.6	Splitting transactions	71
7.7	Pre-allocation of records	74
7.8	Archival	76
8	Limitations of the Diagramming Tool	80
9	PODADE - A Computer-based Aid to Database Design	82
9.1	Architecture and data structures	82
9.2	Brief logic	82

10 Summary and Future Work	85
A Summary of Problems and Solutions (Schema Design)	90
B Summary of Problems and Solutions (Transaction Design)	92

List of Figures

1.1	The Performance Drawbacks of Third Normal Form	2
4.1	Horizontal Partitioning	18
4.2	Personnel Queries	19
4.3	Finance and Accounts Queries	20
4.4	Project Managers' Queries	20
4.5	Ideal Partitioning of Employee Table	21
4.6	Naive Partitioning of Employee Table	22
4.7	Logic of Sequence Number Generator	23
4.8	Solution to the Sequence Number Generation Problem	25
4.9	Solution to the Sequence Number Generation Problem - cont'd	26
4.10	Departmental Managers' Query	26
4.11	Reservation Counter Queries	28
4.12	A Relation and its Index	29
4.13	Stock Exchange Transaction	31
4.14	Ideal Sequence Number Generation Logic	33
4.15	Ideal Sequence Number Generation Logic - cont'd	34
4.16	Sequence Number Generation in Two Transactions	35
4.17	Sequence Number Generation in Two Transactions - cont'd	36
4.18	Sequence Number Generation in Two Transactions - cont'd	37
4.19	One Possible Logical Schema for Containers and Their Locations	38
4.20	Another Possible Logical Schema for Containers and Their Locations	39
4.21	An "Insert/Delete" Approach to Relation Usage	39
4.22	An "Update-only" Approach to Relation Usage	40
4.23	Operations vs. MIS Queries	42

5.1	Basic Notation - Modified ERD	45
5.2	The urge to merge, a graphical illustration	46
5.3	Basic Notation cont'd	48
5.4	Transaction Notation	50
5.5	Basic Notation cont'd	51
5.6	A Join	52
7.1	High Insert Load and Subsequent Contention for the Last Page	62
7.2	Vertical Partitioning based on Attributes Accessed	64
7.3	Sequence Number Generation - a Naive Approach	66
7.4	Sequence Number Generation - Towards a More Efficient Solution	67
7.5	Sequence Number Generation - Redundant Storage	68
7.6	The Case for Clustering	70
7.7	Stock Trading with High Locking Contention	72
7.8	Reading without Locks to reduce Locking Contention	73
7.9	Sequence Number Generation - Transaction Splitting	75
7.10	An "Insert/Delete" Approach to Relation Usage	77
7.11	An "Update-only" Approach to Relation Usage	78
7.12	Operations versus MIS	79
9.1	ERD of the Design Tool	83

Chapter 1

Introduction

1.1 Traditional database design

The automatic design of relational databases is a well-researched area. Algorithms and techniques exist to convert conceptual schemata (EER diagrams) into logical designs (relations in Third Normal Form) [BATI92, ELMA89, RAM 89]. However, database designers find that such a textbook approach to design, while resulting in a system that meets its functional requirements, often fails to provide good performance.

The Entity-Relationship approach to database design is no doubt a popular one among database designers for its simplicity, power of expression and for its remarkable property of leading to relations that are in Third Normal Form. However, Third Normal Form, in spite of its highly desirable properties in terms of freedom from anomalies, is often found to be inefficient in production environments. (See Figure 1.1 for an example). For this reason, database designers talk in terms of "controlled redundancy" to trade off some safety (from inconsistency) for better performance.

1.2 The performance factor

More sophisticated tools and techniques also recommend physical designs (indexes) based on relation cardinalities, attribute value distributions, query frequencies and selectivity factors [ANTO85, CHO 89, FINK88, LEON81, WHAN81]. This is a further step towards generating designs that yield good performance in addition to modelling the application correctly.

Consider a company database which stores information on employees as well as their promotion histories. Clearly, the relationship between the entity Employee and the entity Designation in this case is 1-to-N, since each employee could have held many designations in the course of his or her career. The Third Normal Form representation of this would result in a relation called Employee with attributes like Employee Number, Name, Date of Birth, Address, etc. The data on promotions would however have to go into another relation called Promotion History. This relation would have the attributes Employee Number, Designation, Date Promoted, etc.

While this is a clean and straightforward way to store information, it is quite inefficient at answering queries like "List out all the employees along with their *current* designations". Such a query is likely to be quite frequent. Even worse is the query "List out all the employees who are *currently at this particular designation*". Clearly, it is better to store the current designation *redundantly* in the Employee relation, even if this carries with it the risk of update anomalies.

Figure 1.1: The Performance Drawbacks of Third Normal Form

However, all the approaches so far have certain drawbacks.

1. They inadequately examine (if at all) concurrency and locking issues.
2. They usually do not recommend changes to the logical schema or to transactions, and over-emphasise the importance of index selection.
3. Those that analyse queries and their frequencies consider them as independent statements and not as part of transactions.
4. Some of the approaches to vertical partitioning can aggravate concurrency-related problems.
5. Some also assume that the primary index is built on the primary key, which is not always true.

These points are elaborated in a later chapter.

Some significant problems (other than lock contention) contributing to poor performance are paucity of main memory resulting in excessive paging, fragmentation of data on disk leading to large seek times for queries, etc., but while these may even be the major problems in single-user systems, they are usually insignificant compared to the lock contention problem in multi-user systems. Indeed, when they do exist, they only serve to *aggravate* lock contention (because a slower transaction makes others wait longer). Hence, an attempt to reduce lock contention usually requires one to solve these other problems also. This is a comprehensive, "top-down" approach which is more suitable to performance tuning in multi-user systems.

However, most design techniques that are in common use implicitly assume a single user for the envisaged system, even when they consider all the queries that are to be run. Most testing procedures are also oriented towards checking for the existence of functional features or the logical correctness of programs. There are hardly any design tools which automatically lead to efficient operation in multi-user environments. Performance testing procedures are also unfortunately not general-purpose. Each application demands a customised testing procedure.

At any rate, what designers would like is a tool or technique that would allow them to develop an application keeping performance in mind *right from the beginning*. When a

system is implemented, it should not be necessary to rework large parts of it for performance reasons. Hence even good testing procedures are not enough. It is design techniques that need to be improved.

There are methods to improve the performance of a database application, not all of which are related to data structures and thereby to the Entity-Relationship approach. These include

- The use of secondary indexes to speed up access to records,
- Clustering of data to speed up the selection of *groups of* records,
- Query optimisation (not under the designer's control),
- Partitioning of relations (horizontal and vertical),
- Storage of derived data (redundancy),
- Reduced consistency levels, i.e. reading without locks, splitting transactions, etc.

These help to improve performance under conditions of concurrent access and are frequently used.

Unfortunately, there has so far been no integrated methodology to consider all these factors and make a comprehensive set of recommendations for high performance. The Entity-Relationship approach, which traditionally forms the basis for relational database design, is of course inadequate for this purpose, since it models too few of the factors that affect performance.

1.3 A new attempt

As reported in [GCP93a], we surveyed some RDBMS application sites, studied their performance problems, and noted the decisions taken by designers to solve those problems. We found that the major performance problems were caused by lock conflicts between concurrent transactions. The solutions to these problems ranged from the obvious to the innovative (See Appendices A and B). We had then expressed the need for a design tool that would automatically analyse situations for performance bottlenecks like an experienced human designer would, and suggest suitable solutions.

We now present such a design tool in the form of a diagramming technique [GCP93b]. By following some well-defined steps, we show that it is indeed possible to systematically derive the solutions to most of the problem situations described in the earlier paper.

Our target system is an RDBMS-based application in the "mid- range", i.e. neither a trivial system for one or two users nor a major (say) stock exchange application with a very high transaction rate and which probably needs specially-tuned hardware and system software. We believe, from our observations of some existing applications [CMC 93], that most sites (80-20 rule?) exhibit database sizes in the range of 100 MB to 1 or 2 GB, have 10-30 concurrent users, and have "moderate" transaction rates.

This is of course not a rigorous classification. What we intend is to eliminate from consideration those systems where the application is either so trivial as to survive even inefficient designs or so complex as to necessitate special hardware or system software.

Throughout this paper, we will use the term "record" interchangeably with "tuple" and "table" interchangeably with "relation".

Chapter 2

Inadequacies of Earlier Approaches

2.1 Why concurrency and locking are important

Most techniques, even when they consider transaction volumes, often implicitly assume a single user. What they seek to minimise are the time taken to perform certain activities (eg. disk seek time, data transfer time) and other factors like the number of blocks accessed.

But in real multi-user systems, contention among users is the predominant cause of poor performance. Further, *data contention* is often more critical to performance than *resource contention* ([AGRA85] report that even with infinite resources, the throughput of a system with blocking-based concurrency control starts going down when the degree of multiprogramming crosses a certain value). To take an extreme example, an RDBMS-based system running on a Cray supercomputer with only two concurrent users could still show poor performance if one of the users was browsing through a relation with read locks on all pages. The other user would then be unable to update even a single record in that relation and would wait indefinitely. The paradox is one of poor performance in spite of a "low" transaction rate and plentiful resources.

Hence we believe that any realistic design must take data (lock) contention into account. It is not enough to say, for example, that there are 3 types of transactions T1, T2 and T3 and that their frequencies are 100/day, 500/day and 300/day. We need to know the pattern of access conflicts that occur between them.

2.2 Why the design of logical schema and transactions is more important than choosing an index configuration

Database design tools deal mainly with physical database design (i.e. choice of primary and secondary indexes). Sometimes, vertical partitioning is also considered at the logical level [BATI92, ROZE91, ANTO85, NAVA89, HAMM79]. In practice, application implementors and database administrators find that major improvements in performance come from changes to the *logical schema* (i.e. horizontal and vertical partitioning, redundant storage), from changes to *system parameters* (eg. buffer space) and from *transaction redesign*. Changes at the physical level alone are found to be inadequate in many cases [CMC 93].

While a good index configuration speeds up performance significantly, this is not as crucial a decision as one might think, because the cost of rectifying a mistake is not too high. Indexes can be created and dropped on the fly in most RDBMSs. In contrast, a change to the logical schema (after an application has been developed) involves changes to all programs that use those relations and a significant amount of data unload/reload. Even in an RDBMS like Oracle, where clustering a relation is difficult - first create a cluster, then the cluster index, and finally load the data - it is still far easier than making changes to the logical schema after implementation.

An application designer would be more worried about design decisions that are difficult to change subsequently, and a tool that helped in efficient logical schema design and transaction design would be more welcome than one which suggested only an index configuration (which could in the worst case, be arrived at through trial and error without changing the application in any way).

2.3 Why transactions are more important than individual query statements

Many tools consider each SQL statement independently. Their emphasis is on weighting each SQL query by its frequency in order to arrive at a globally optimal configuration of indexes, and they neglect the more important effect of lock conflict when a *group of such statements forms a single transaction*. All locks acquired during the execution of

statements within a transaction are held for the entire duration of the transaction, and are released only when the transaction as a whole commits. This increases the time for which each relation is locked and thereby the probability of lock conflicts between transactions.

We have also observed that many existing applications exhibit a fairly sophisticated schema design, while their transactions show plenty of scope for improvement. Probably this is because schema design is done by experienced analysts, while programming is often left to more junior persons in many organisations. Not all the inefficiencies introduced by these programmers are caught and rectified later during testing and walk-throughs, because these are not mistakes, only inefficiencies.

We therefore believe there is a need for a tool that recommends more efficient transaction designs. For example, it should point out where reading can be done without locks, and where transactions can be split without the danger of inconsistency.

2.4 How vertical partitioning may sometimes aggravate performance problems

The rationale for vertical partitioning is clear. In a relation, if there are two or more groups of attributes that are *frequently jointly accessed*, but each group is *accessed independently* of the others, then it makes sense to place each such group in a separate relation with the key repeated.

The key phrases are "frequently jointly accessed" and "accessed independently". This is what clusters a set of attributes together and separates it from other sets.

In two techniques [ROZE91, ANTO85], vertical partitioning is done differently, by separating attributes that are *frequently accessed* from those that are *less frequently accessed*. The rationale for this is that separating out the most frequently accessed part of a relation makes it smaller, and since more records now fit into a page, fewer block accesses are needed to fetch a given number of records.

However, this technique reckons without lock contention. Since more records now fit into a page, and since all frequently accessed attributes are now concentrated in a single, smaller relation, the probability of lock conflicts *increases*, especially in page-level locking systems. Thus, this naive approach to vertical partitioning may actually make performance worse.

Even in the case where partitioning is done by separating groups of frequently-accessed attributes based on the transactions that access them [NAVA89], there could possibly be increased contention in each smaller table because each *type* of transaction might still be run by multiple users. This increased contention in page-level locking systems seems to be an unavoidable side-effect of vertical partitioning. So in addition to partitioning, we might need to do something more, perhaps reduce the fill factor of data pages to decrease the number of records per page.

2.5 Why a primary index need not be built on a primary key

[ANTO85] assume that a primary index is built on the primary key of each relation, and confine their attention to secondary index selection. [FINK88] do not make this assumption, and we agree with them for the following reasons:

A primary key is an aspect of *logical* database design. The primary key is an attribute (or a set of attributes) that uniquely identifies a record. It would come in useful when the requirement is to locate a *particular* record based on that attribute value.

A primary index is an aspect of *physical* database design. A primary index on an attribute physically sorts the relation on that attribute (When the attribute's values are non-unique, a primary index is called a cluster index). Physical sorting and clustering expedite range queries and group selects, because the desired records are all found in close proximity. Here, merely being able to *locate* records (as with the help of a secondary index) is not sufficient, since the records may be scattered over many physical blocks requiring extensive disk arm movement.

Now why should a relation be sorted on its key value? Selection based on the key returns a single record (by definition). Sorting and clustering do not improve performance when a *single* record is to be selected. In such cases, a secondary index is sufficient to speed up the process of locating the record. It is only when a set of records is to be selected *ordered by the primary key* that we might need a primary index on the primary key, and such queries are rare [CMC 93].

To drive this home with an example, a relation holding employee data as shown below would probably have a *primary* (cluster, actually) *index* on a *non-key attribute* like

Department, and a *secondary index* on the *primary key* - Employee number.

(Employee number, Name, Date of birth, Department, ...)

There are probably two kinds of transactions that access this relation. Some access single records based on the key (as in "Get me all the details of this employee"), and some access a group of records based on a non-key value (as in "Get me the details of all employees in *my* department").

Clearly, a secondary index on Employee Number is sufficient to locate a *single* record as required by the first kind of query. The cluster index on Department not only reduces block accesses for the second kind of query, it reduces *lock contention* between users who may be querying the same table for details of employees belonging to other departments.

In our experience, a primary index is rarely built on a primary key.

Chapter 3

Approach of This Thesis

3.1 Solving the real problem

Most current design tools look at performance from a limited viewpoint. They concentrate on improving the efficiency with which *individual* queries are executed. If there are conflicts between two queries in terms of the strategies needed to speed them up (eg: We can't have two different primary or cluster indexes on the same relation), the less frequent one is sacrificed for the other. Different costs of access are assigned to different (say) index configurations, and the least cost configuration is selected, *even if that cost is still high in absolute terms*.

Our approach is more comprehensive and subsumes the traditional one. We look at the *ideal* multi-user system as one which guarantees absolute parallelism of transactions. This is the situation that provides maximum possible throughput. At the other extreme is a batch kind of system that strictly serialises transactions so that no two of them can execute concurrently. Most concurrency control mechanisms used in relational databases result in a system that lies somewhere between absolute parallelism and strict serialisation. In other words, concurrency control results in some queueing. Since our ideal system is a completely parallel one, we must try and reduce unnecessary queueing.

[GRAY78] refers to an analysis which concludes that the probability of deadlock rises with the square of the degree of concurrency and with the fourth power of the transaction size. This implies that lock conflict in general (of which deadlock is a subset) must also increase at this rate *at least*. We must therefore look at ways to reduce lock conflict by reducing the size of transactions, the duration during which locks are held and the degree of

concurrent access to database objects. The techniques we describe in the coming chapters aim to do one or more of these.

In general, the response time of a query (or transaction) is affected more by the time it has to wait in a queue behind another query (or transaction) than by the time the system takes to execute it. This can be shown very simply. If there is a queue of a fixed length and all transactions take the same time to execute, the average waiting time of a transaction will be proportional to the average execution time for such transactions. Further, the longer the queue, the greater the ratio of waiting time to execution time for any query or transaction. For any queue of length greater than one, the waiting time is at least as much as the execution time.

So reducing the processing time is important for improving the response time of a transaction, not so much because the time taken to execute *it* is reduced, but because the time taken to execute *other* transactions in the queue ahead of it is reduced, and thereby the waiting time of *this* transaction.

We present below some examples which prove that this approach (of reducing unnecessary queueing) is richer than the traditional one (of speeding up individual transactions).

3.2 Two ways to make a queue move faster

Consider the queue at a library check-out counter. Assume that borrowers arrive at the rate of one a minute. Also assume that the counter clerk disposes of each borrower at the rate of one a minute. It is easy to see that there is a queue of only one person at a time, and there is no waiting time for any person. On the other hand, if the counter clerk is replaced by an apprentice who takes two minutes to dispose of a borrower, then it is easy to see that the queue starts building up at the rate of one more person every two minutes.

One way of solving the problem is to make the process faster, i.e. replace the novice with a faster counter clerk. But there is another way. We can open another counter, maybe even have another apprentice there. We haven't made the first process faster, but we have solved the *queueing* problem and thereby improved the throughput and the response time. This second kind of solution occurs to us only if we look at the problem as one of unwanted serialisation. We then think of ways to shorten the queue and come up with at least two different methods.

Adopting the approach of merely speeding up transactions that seem to be slow may give us some solutions but not others. In a database case, trying to speed up a query may lead a designer to build an index. Trying to reduce queueing, on the other hand, may suggest partitioning (similar to having a second library counter) *as well as* building an index.

3.3 A design technique that neglects queueing - and pays for it

As we discussed in a previous chapter, many design tools and techniques approach vertical partitioning by dividing the attributes of a relation into two groups - frequently accessed and infrequently accessed, and putting each group into a separate relation. The rationale for this is as follows.

A block of storage (which usually maps to a page of memory) can hold only a certain number of bytes. If a record is many bytes wide, then only a few records will fit into a block. If the record is smaller, then more records can be accommodated within it. Since a block is a unit of I/O, we would like records to be selected in as few block I/Os as possible. So we would like to pack as many records as possible into a single block.

On the other hand, a page is a unit of locking in many systems. Whenever a record is selected, the page on which it lies is locked, thereby locking even those other records on that page which may not be required. Since we are sensitive to locking and the resulting serialisation of transactions that results from it, we would like to lock as few unwanted records as possible. This consideration leads us to desire exactly the opposite situation, i.e. to have as few records in a page as possible.

The apparent contradiction is resolved when we realise that access by many concurrent users to sets of records is expedited by dense packing (especially if the sets are disjoint), and to single records by sparse packing. Further, if different types of transactions access different sets of attributes, then vertical partitioning into that many relations is called for, since transactions of each *type* then form separate queues. In fact, vertical partitioning is the best strategy in the absence of attribute-level locking.

As we pointed out earlier, the approach of partitioning attributes into frequently- and infrequently-accessed ones may actually make performance worse, because queue lengths at each page may *increase*.

3.4 Clustering is better than you may think

One of the well-known effects of clustering the records of a relation together on an attribute is that when all records with a particular value of the attribute are selected, or when records are selected one by one ordered by that attribute, the records will all be found in contiguous physical locations. Disk I/O will therefore be faster, transactions will get completed faster, and waiting times for other transactions will be reduced, enabling *them* to complete faster. Indeed, this reasoning is used in all the index selection algorithms surveyed.

But clustering can also have another beneficial side-effect that these techniques do not consider. If concurrent transactions access different portions of a relation based on different values of the same attribute, then clustering the records based on that attribute will not only *group* the records accessed by a transaction together, but also *separate* such groups from one another. Especially in page-level locking systems, it improves performance tremendously if each transaction only locks the records it needs.

Again, we can see that the solution was to break up a single queue at a relation into several queues, each at a different set of pages.

With these examples, it is clear that an approach based on reducing unnecessary queuing can lead to better solutions for performance than those followed so far.

3.5 Methodology of work

1. We studied 5 RDBMS application sites in detail and interviewed systems analysts and database administrators to get an idea of what performance problems usually occurred in real-life DBMS applications. We noted the techniques employed by them to solve such problems and analysed these solutions to extract their basic principles, if any.
2. We studied the literature extensively to get an idea of how current design tools and techniques approach the problem.
3. We found a mismatch between the problem as perceived by application implementors in the field and as perceived by the design approaches we studied.
4. We then set about developing a design technique of our own that would address the real problem that application implementors were trying to solve.

3.6 Contribution of this thesis

1. The major contribution of this work is that it is probably the first design technique to identify the real problem, i.e. lock contention in multi-user systems, as the bottleneck to be broken.
2. The second contribution is the development of an intuitive, diagrammatic technique that helps a designer model data as well as transactions in one integrated notation. This technique makes conflict areas stand out visually. Even potential deadlocks can be detected in many cases by an examination of the diagram.
3. Another important contribution is the compilation of a set of heuristics (rules of thumb) used by experienced designers to ease bottlenecks and improve concurrent performance.
4. Finally, the viability of the method has been proved through the development of a prototype design tool that runs on a computer, interactively acquires all relevant information, and helps a human designer reach a demonstrably more efficient design.

Chapter 4

Case Studies

Let us now consider some cases of poor performance from real-life situations and the solutions that were actually adopted to solve them. These solutions were arrived at either through trial-and-error, "intuition" or "experience". No formal technique was used. In a later chapter, we will apply our diagrammatic technique to these very cases and attempt to systematically derive solutions to them.

For the purpose of explaining the use of different techniques, we present several cases in order of solution type, and describe the problem which each could solve.

4.1 Schema (Logical and Physical) design decisions

Many of the design decisions that can be taken based on the factors we describe are at the schema level, i.e. they deal with the formats of relations, their storage structures and access structures.

4.1.1 Horizontal partitioning

Consider a situation where many concurrent users are trying to insert records into the same relation. Examples could be stockbrokers trying to enter their bids into a "bid" relation, financial operations inserting records into a ledger relation, transactions being logged in a "log" relation, etc. In all cases, the contention between users is for exclusive locks on the *last page* of the relation (in page-level locking systems). There is a limit to how short the transaction can be. Indexes, if they exist, are of no use to an insert operation. In fact, they only make the operation slower because the index itself has to be updated. If B-tree

cluster indexes are used so that the inserts are not always to the last page, in addition to the earlier problem of index locking and update, the structure gradually degrades and the index has to be rebuilt anyway [HABE90].

This leads to a "catch-22" where we cannot afford to have any structure for the relation other than "heap", and such a structure leads to unavoidable contention for the last page.

One possible solution could be horizontal partitioning. In the stock exchange example, we could have one table per stock, greatly easing the load on a common "bid" table. In the financial application, each type of transaction (vendor bills, administrative expenses, payroll, etc.) could insert records into a separate table, thus spreading the insert load across many relations and reducing the contention for locks [CMC 93].

The mapping of transactions to tables could be static or (partially) dynamic. In the financial application, one table could be dedicated to each type of transaction. In the stock exchange case, one table could be dedicated to each heavily- traded stock, and less heavily-traded stocks could share tables. Since the pattern of trading changes gradually with time (a "hot" stock three months earlier may be "sluggish" today), the mapping from stock to table should not be completely static. Figure 4.1 illustrates how partitioning may be done in this case.

This scheme may require the maintenance of a "mapping table" as shown in Figure 4.1, which says which stock should go into which bid table. Though every transaction now has to read this table, this is NOT a bottleneck because all transactions are only readers, so lock contention does not arise. The mappings are changed only rarely, when changes in trading patterns emerge, so the table can be treated as read-only. Access to this mapping table will also be quite fast since it is small and likely to be found in the buffer pool.

Some RDBMSs support horizontal partitioning at the *physical* level. There is only one table at the logical level (with only one name), but this is stored on several disk partitions and the records on different partitions do not map to the same set of pages. If the DBMS supports partitioning based on either attribute values or by a round-robin scheme, this could effectively reduce lock contention. If the DBMS does not support horizontal partitioning at the physical level, the designer has to do it at the *logical* level, i.e. have more than one table (each with a different name) and modify the programs to recognise these tables and choose the appropriate one.

Thus, in an apparent paradox, operations can be made faster by adding an extra step,

Before horizontal
partitioning

```
/*
Insert into
bid table
*/
EXEC SQL
INSERT INTO Bid
(...)
VALUES
(...);
```

After partitioning

```
/*
Find the bid table
for this stock
*/
EXEC SQL
SELECT TableName
INTO :TableName
FROM MapTable
WHERE StockCode = :StockCode;
```

```
/*
Insert into appropriate
bid table
*/
EXEC SQL
INSERT INTO :TableName
(...)
VALUES
(...);
```

Figure 4.1: Horizontal Partitioning

```

SELECT EmpNo,                               SELECT Grade, PromDate
      EmpName, PromDate                     FROM   Employee
FROM   Employee                             WHERE  EmpNo = :EmpNo
WHERE  Grade = :Grade
ORDER BY PromDate

UPDATE Employee
SET    Grade      = :NewGrade,
      PromDate    = :PromDate,
      CurBasic    = :NewBasic,
      CurAllow    = :NewAllow
WHERE  EmpNo = :EmpNo

```

Figure 4.2: Personnel Queries

if as a result of that step, contention for locks goes down.

4.1.2 Vertical partitioning

Consider an MIS application where the Employee table has attributes relating to Personnel, Finance and Projects. The record structure might look like this:

(Employee number, Name, Address, Department, Date of birth, Current Grade, Date Promoted, Current Basic Pay, Current Gross Allowances, Current Project, Date of joining Project)

The primary key is Employee number.

Users in the Personnel department are interested in drawing up lists of people in each grade in order of seniority. They also want to know the seniority of specific employees. When intimated by an employee's manager, they also update his/her grade and the pay and allowances. These queries are shown in Figure 4.2, expressed in SQL.

Users in the Finance and Accounts department are interested in the amount of money spent on salaries and allowances in each department. The SQL statement corresponding to

```

SELECT DeptCode,
        SUM( CurBasic ),
        SUM( CurAllow )
FROM    Employee
GROUP BY DeptCode;

```

Figure 4.3: Finance and Accounts Queries

SELECT EmpNo, EmpName,	UPDATE Employee
FromDate	SET CurProj = :NewProj,
FROM Employee	FromDate = :SinceDate
WHERE CurProj = :MyProj	WHERE EmpNo = :EmpNo

Figure 4.4: Project Managers' Queries

this query is shown in Figure 4.3.

Project managers want to know which employees are currently working on their projects and since when. In collaboration, they may shift an employee to another project. Figure 4.4 illustrates these operations.

The Address attribute is rarely accessed.

We approach this problem with the objective of reducing lock contention, not of merely reducing block accesses. It may then be desirable to split the Employee table above into three tables as shown in Figure 4.5.

The Address attribute may be stored with any of these tables. Each of these is accessed by a different set of users and lock contention on the common table is reduced. Of course, this partitioning is not without its costs. It results in Personnel having to access two tables to update grade, pay and allowances. Project Managers have to access two tables through a "join" to retrieve employee names and project details.

Now we could either live with the cost of the join, or store Employee Name redundantly

EmpGrade:

(Employee number, Name, Current Grade, Date of Promotion)

EmpPay:

(Employee number, DeptCode, Current Basic Pay,
Current Gross Allowances)

EmpProj:

(Employee number, Current Project, Date of joining Project)

Figure 4.5: Ideal Partitioning of Employee Table

in the project table also and avoid the join. Since the name of the employee is rarely changed, no anomalies can result. But the joins, while taking more time than selects from a single table, do not necessarily increase lock contention. If partitioning results in all transactions needing to "join" with a common table, but that table is never updated, then lock contention actually *goes down*, because readers never block readers. We could, if required, make the joins faster by building appropriate indexes.

Updating two tables instead of one (as in the case of Personnel) is a more serious matter. If this is a frequent query, we may have to merge the two tables.

To reduce the increased contention among *similar* transactions (eg. three concurrent users from Personnel) as a result of this partitioning, we may decrease the fill factor of the data pages where each such relation is stored so that we have fewer records per page [CMC 93].

Contrast this with the approach of merely dividing the attributes into "frequently accessed" and "infrequently accessed" without regard to the transactions that access each attribute. The partitioning that then emerges is shown in Figure 4.6.

EmpMain:

(Employee number, Name, Date of birth, Department,
Current Grade, Date Promoted, Current Basic Pay,
Current Gross Allowances, Current Project,
Date of joining Project)

EmpAux:

(Employee number, Address)

Figure 4.6: Naive Partitioning of Employee Table

Since Address is likely to be a fairly large field, the removal of this will make EmpMain a significantly smaller table than Employee. The same (conflicting) transactions now end up accessing a smaller table, so contention only gets worse.

4.1.3 Redundancy

Consider an application where new records are being inserted (identified by running sequence numbers) by many users into the same relation. Each user must first search the relation for the last sequence number used, then insert a record with the next number in sequence. However, this approach suffers from two major problems. The scan of the relation takes up a lot of time, increasing the time for which the transaction holds its locks (It is not advisable to build an index on this relation because of the heavy insert load). There is a high probability of deadlocks because two transactions can simultaneously read the relation, but when they try to update, each has to wait for the other to relinquish its locks.

(We assume that the sequence number is not just a number, but some set of characters followed by a number. The overall code has to be unique. In an application registering investors, for example, the first investor with initials ASR gets the number ASR0000001,

```

read( Initials );

EXEC SQL SELECT MAX( SeqNo )    /* Aggregation */
        INTO      :SeqNo
        FROM      Investor
        WHERE     Initials = :Initials;

if ( sqlcode == 100 ) /* No record found */
    SeqNo = 1;
else
    SeqNo++;

EXEC SQL INSERT INTO Investor
        ( Initials, SeqNo, ... )
VALUES
        ( :Initials, :SeqNo, ... );

EXEC SQL COMMIT;

```

Figure 4.7: Logic of Sequence Number Generator

the next investor with initials (say) PGD gets the number PGD0000001, the next investor with initials (say) ASR again, gets the number ASR0000002, etc. This cannot be served through a DBMS-provided sequence number generator, since the number of unique initials increases with time [CMC 93])

Figure 4.7 illustrates this logic.

One solution which avoids deadlocks is where each transaction takes an exclusive, table-level lock on the Investor relation before proceeding. But the time taken to scan the relation still means that the transaction duration is long, increasing lock contention. This problem can be solved by storing the latest sequence numbers for each unique set of initials

redundantly in a separate table with an index on the attribute Initials [CMC 93]. Reading this table takes very little time because the index obviates the need for a scan. Another reason why it takes less time is that the table is smaller, and the most frequently-accessed pages are usually found in the common buffer pool. Each transaction therefore has only to take an exclusive lock on this second relation, read the latest sequence number, increment it by one, update or insert the latest sequence number's value into this second relation (which usually doesn't affect the index because inserts are rare) and insert a new record with this number into the original table as described in Figures 4.8 and 4.9.

(In cases where sequence numbers may be skipped without affecting the correctness of the application, this transaction could even be split, as detailed later).

[PEIN88] also describe a stock exchange situation where a transaction (on inserting a new bid) has to search the entire bid table to find if a matching bid exists. They report that this search is avoided by storing the highest buy price and the lowest sell price of a stock *redundantly*. Comparison of a new bid's price to the current price range is then far faster. In most cases, no deal is possible, so a wasteful scan is avoided. Only when the price range indicates that a match may exist does a full scan become necessary.

In general, when queries require to scan entire relations or large parts of them to get at some aggregate value (through COUNT, SUM, MIN, MAX or AVG), and when other transactions want to update parts of the same relation, conflicts are likely because the read operation takes a long time and locks a large number of pages. Redundant storage of information is a solution in all these cases. Care should however be taken in designing transactions so that anomalies do not creep in.

4.1.4 Clustering

We often have situations where each user accesses a *subset* of a relation, and *the subsets are all disjoint*. The example we saw earlier of departmental managers making queries regarding the employees of their own departments falls into this category (Figure 4.10).

Clustering the Employee relation on the DeptCode attribute makes good sense even in the single user case since it reduces the number of accesses needed to fetch the desired records. It makes even better sense when we consider concurrency.

Assume an Employee relation with 10,000 records. Let there be 10 departments with around 1000 employees each. Let the record width be 250 bytes and the size of a page be

```

EXEC SQL SET LOCKMODE ON LastSeq
        WHERE READLOCK = EXCLUSIVE;

read( Initials );

/* Fast select using index on Initials */
EXEC SQL SELECT LastSeqNo
        INTO      :SeqNo
        FROM      LastSeq
        WHERE     Initials = :Initials;

if ( sqlcode == 100 ) /* No record found */
{
    SeqNo = 1;      /* Affects index, but rare */
    EXEC SQL INSERT INTO LastSeq
            ( Initials, LastSeqNo )
            VALUES
            ( :Initials, :SeqNo );
}
else
{
    SeqNo++;      /* Doesn't affect index */
    EXEC SQL UPDATE LastSeq
            SET      LastSeqNo = :SeqNo
            WHERE     Initials = :Initials;
}

```

Figure 4.8: Solution to the Sequence Number Generation Problem

```
EXEC SQL INSERT INTO Investor
    ( Initials, SeqNo, ... )
VALUES
    ( :Initials, :SeqNo, ... );

EXEC SQL COMMIT;
```

Figure 4.9: Solution to the Sequence Number Generation Problem - cont'd

```
SELECT EmpNo, EmpName, ...
FROM Employee
WHERE DeptCode = :MyDeptCode
AND ...
```

Figure 4.10: Departmental Managers' Query

2000 bytes. Then 8 records fit in a single page. The entire relation occupies 1250 pages. If the records are spread *randomly* over all these pages, in the worst case, the records pertaining to any single department may be spread over 1000 pages, i.e. one record per page. Then any two simultaneous queries will definitely have at least one page in common. If the lock modes are incompatible, this will end up strictly serialising the transactions. If on the other hand, it is known that a user will access only the records pertaining to his own department, then clustering the relation on Department will lead to the records of a department occupying only 125 pages. In addition to making group selects faster (125 pages as opposed to 1000 pages), the records of one department have no pages in common with those of another. Hence lock conflicts never occur and we have complete concurrency.

In practice, there is usually an overlap between records of "adjacent" departments, leading to some conflict, but this level of conflict is far lower than in the case where no clustering exists.

Most current design techniques require the designer to state the transaction frequencies as well as describe the query itself and its estimated selectivity. In the above case, as we can see, merely stating the query and assigning a value of 0.1 to the selectivity does not help us make this clustering decision. We need to know that each user accesses a *different* fraction of the table. In other words, it is the *degree of overlap* between the subsets accessed that determines the probability of conflict and the means of resolving it.

We could consider a very similar case of a relation with 10,000 records and 10 concurrent users, each accessing 1000 records. In this case, however, the records refer to some financial data of the last 10 months, from January to October, with 1000 records per month. It is likely that all the 10 users are interested in the *latest* data, i.e. of October. So though the relation cardinality (10,000 records), the number of concurrent users (10), the "where" clause of each user and the selectivity of each query (0.1) are similar, clustering does not help us in this case. We may have to resort to redundant storage if some of the queries are aggregations. As we can see, the usual parameters collected by current design tools are not enough for us to make a good decision for performance. We need to be able to determine the level of conflict between transactions in the system.

SELECT Status	UPDATE Bookings
FROM Bookings	SET Status = 'Confirmed',
WHERE ResSeqNo = :ResSeqNo	SeatNo = :SeatNo
	WHERE ResSeqNo = :ResSeqNo

Figure 4.11: Reservation Counter Queries

4.1.5 Hashing/De-clustering

Another pattern of access is observed when concurrent users access single records each (possibly not the same records), but the records in some sense are logically grouped together. For example, in railway reservations, there might be some identifying "reservation sequence number" assigned to each ticket (independent of seat number). This implies that tickets booked at roughly the same time have sequence numbers close together. Queries regarding confirmation are usually on this sequence number, and it is likely that records pertaining to tickets booked on a particular day for a particular train will lie close to each other. This kind of access pattern could lead to a lot of lock contention in page-level locking systems, since each query accesses a single record, but locks a page, possibly locking adjacent records required by other transactions. Examples of operations are shown in Figure 4.11, expressed in SQL.

One solution which we discussed earlier was a lower fill factor for the data pages, so that fewer unwanted records are locked by a transaction. Another technique is hashing the records by sequence number value. Then *logically* adjacent records will not be *physically* adjacent. When the hashing is done across disks, this is also known as de-clustering.

Basically, this technique *converts a skewed access to a uniform one*, and reduces contention. However, this is good only under certain conditions. It could very easily conflict with some other access pattern which dictates clustering (group select) or no structuring at all (heavy insert load).

Table	Index
(Attrib1, Attrib2, ...)	(Attrib2, TupleId)

Figure 4.12: A Relation and its Index

4.1.6 Indexes

Under certain concurrency conditions, a database designer might decide *against* having an index. Unlike a decision on how much buffer space to allocate, a decision on indexes involves trade-offs (More buffer space can never make performance worse. At worst, performance will remain the same. However, adding an index need not always make transactions faster. Some transactions could become *slower* as the result of adding an index).

Since an index consists of (attribute value, tuple id) pairs, whenever a new record with some value for that attribute is added or deleted, the index has to change to reflect it. A change in that attribute value is equivalent to an insert and a delete as far as the index is concerned. These index update costs have been recognised and catered to in tools for index selection. They weigh the benefits of having an index against its update costs and try to arrive at a good, if not optimal set of indexes which are best for a given suite of queries.

But there is another cost of having an index in concurrent update situations. Very often, *the index itself becomes a source of contention*. To see this, consider an index (organised as a B- tree) on an attribute (Attrib2) of a relation. Any update of this attribute (or an insert or delete of a record) will affect the index, because the set of tuples corresponding to this attribute will change (Figure 4.12).

Since the tree may need to be reorganised as nodes split or merge, rather conservative locking schemes have to be adopted. The "deepest safe node" is usually locked before any index update. The large fan-out of the B-tree means that very many leaf level nodes are in effect locked when an ancestor node is locked. Thus, even though two concurrent updates may not conflict in their data pages, there could be a conflict in the index [SRIN91].

It would seem that an application *should go in for fewer indexes than indicated by any current design tool*, because these tools do not consider index locking costs. The worst

effect of not having an index is a sequential scan of a relation, making transactions longer. Sometimes, this cost may be preferred to the cost of suffering index lock contention.

4.2 Transaction and usage design decisions

Designers can derive better performance not only from intelligent schema design but from more efficient access to relations. Some part of the work is done by the DBMS' query optimiser, but major efficiencies can be derived by the designer through smart transaction design. Other decisions (eg. archival policy) can also contribute to efficiency.

4.2.1 Reading without locks

The textbook example of a read that does not require locks is an MIS query. Presumably such queries do not care if data is slightly inconsistent. In addition to this well-known case, there are other situations where reading without locks helps in improving concurrency performance without sacrificing consistency.

[PEIN88] have described a situation at a stock exchange where transactions have to check if their current bid price changes either the maximum buy price of a stock or its minimum sell price. Normally, there are many outstanding bids and the maximum buy price is less than the minimum sell price for any stock, so no deal is possible. If a transaction ever changes these price ranges, a deal *may* become possible. The transaction is described in Figure 4.13.

This holds read locks on the Stock table while the condition is checked, and the table is updated if a certain condition is met. It is observed that the condition is rarely satisfied, so most transactions take locks unnecessarily, delaying the few which really need to update the table.

[PEIN88] report that the processing was changed to first read *without locks*. If the price range has to be changed, the entire operation is *redone with locks*. This technique can be generalised as follows:

When a transaction performs a read, and then performs an update of the value only if the value falls within a certain range, and if the probability of this happening is low, then the transaction can first read without locks. In most cases, the condition will not hold, so the transaction will terminate without having taken any unnecessary locks. If

```

/* Find price range */
EXEC SQL SELECT MaxBuyPrice,
               MinSellPrice
        INTO   :MaxBuyPrice,
               :MinSellPrice
        FROM   Stock
        WHERE  StockCode = :StockCode;

/* Change price range if required */
if ( TransactionType == BUY )
{
    if ( Price > MaxBuyPrice )
    {
        EXEC SQL UPDATE Stock
                SET      MaxBuyPrice = :Price
                WHERE  StockCode = :StockCode;
    }
}
else
{
    /* Sell transaction */
    if ( Price < MinSellPrice )
    {
        EXEC SQL UPDATE Stock
                SET      MinSellPrice = :Price
                WHERE  StockCode = :StockCode;
    }
}

/* Further processing ... */

```

Figure 4.13: Stock Exchange Transaction

the condition turns out to be true, then the transaction repeats the operation with locks. This extra overhead isn't too high overall because of the low frequency of the condition. Reading without locks avoids conflict between unsuccessful transactions (the majority) and successful ones.

4.2.2 Splitting transactions

Since transaction duration is one of the factors contributing to lock conflicts [GRAY78], we must try to split transactions into smaller ones wherever possible, without of course, rendering the database vulnerable to inconsistency. [SHAS92] present an excellent analysis of the situations in which this can be done. However, in most large transactions we have seen (where chopping would potentially yield the greatest benefits), the writes tend to occur towards the end, after the reads. So the condition imposed by [SHAS92], i.e. that no part of a chopped transaction other than the first can have rollback statements, is often not satisfiable.

However, we have seen situations where chopping is acceptable even when this condition is violated. When we split a transaction into two or more parts, we are committing a part of the transaction prematurely in order to avoid holding on to locks for too long. As a result, we lose the ability to roll back an earlier part of transaction if a later part fails. This implies that when the (compound) transaction is retried after a failure, earlier (committed) portions may end up *being redone*. But in many applications, this earlier part may actually be *idempotent*, capable of being run several times without causing inconsistency, or repeated retries may result in changes that do not materially affect the application. To take an example of this latter case, consider the sequence number generation problem discussed earlier. The ideal transaction would look like the one shown in Figures 4.14 and 4.15.

This will result in the generated numbers being strictly in sequence with no gaps. If the application does not require strict sequence as long as the numbers are unique and increasing, then it does not matter if a few numbers are skipped in the middle. The modified transaction would look like the one shown in Figures 4.16, 4.17 and 4.18 [CMC 93].

In this case, it might happen that the *whole* transaction may not roll back on an error that occurs in the final insert statement (say) because the first part would already have committed. Such a situation might result in a sequence number never being used, because the next successful transaction will obtain the *next* sequence number. We then end up with

```

EXEC SQL SET LOCKMODE ON LastSeq
        WHERE READLOCK = EXCLUSIVE;

/* Find the last sequence number for these initials */
EXEC SQL SELECT LastSeqNo
        INTO      :SeqNo
        FROM      LastSeq
        WHERE     Initials = :Initials;

if ( sqlcode == 100 ) /* No record found */
{
    /*
    These initials didn't exist before.
    Add the initials and the number 1.
    */
    SeqNo = 1;
    EXEC SQL INSERT INTO LastSeq
            ( Initials, LastSeqNo )
            VALUES
            ( :Initials, :SeqNo );

    if ( sqlcode < 0 )
    {
        EXEC SQL ROLLBACK;
        return (1);
    }
}.

```

Figure 4.14: Ideal Sequence Number Generation Logic

```

else
{
    /* Increment the last sequence number */
    SeqNo++;
    EXEC SQL UPDATE LastSeq
        SET      LastSeqNo = :SeqNo
        WHERE    Initials = :Initials;

    if ( sqlcode < 0 )
    {
        EXEC SQL ROLLBACK;
        return (1);
    }
}

/* Use the sequence number just obtained */
EXEC SQL INSERT INTO Investor
    ( Initials, SeqNo, ... )
VALUES
    ( :Initials, :SeqNo, ... );

if ( sqlcode < 0 )
{
    EXEC SQL ROLLBACK;
    return (1);
}

EXEC SQL COMMIT;

```

Figure 4.15: Ideal Sequence Number Generation Logic - cont'd

```

EXEC SQL SET LOCKMODE ON LastSeq
        WHERE READLOCK = EXCLUSIVE;

/* First part of transaction begins */

/* Find the last sequence number for these initials */
EXEC SQL SELECT LastSeqNo
        INTO      :SeqNo
        FROM      LastSeq
        WHERE     Initials = :Initials;

if ( sqlcode == 100 ) /* No record found */
{
    /*
    These initials didn't exist before.
    Add the initials and the number 1.
    */
    SeqNo = 1;
    EXEC SQL INSERT INTO LastSeq
            ( Initials, LastSeqNo )
            VALUES
            ( :Initials, :SeqNo );

    if ( sqlcode < 0 )
    {
        EXEC SQL ROLLBACK;
        return (1);
    }
}

```

Figure 4.16: Sequence Number Generation in Two Transactions

```

else
{
    /*
    Increment the last sequence number
    */
    SeqNo++;
    EXEC SQL UPDATE LastSeq
           SET    LastSeqNo = :SeqNo
           WHERE  Initials = :Initials;

    if ( sqlcode < 0 )
    {
        EXEC SQL ROLLBACK;
        return (1);
    }
}

/*
Commit and release all locks
*/
EXEC SQL COMMIT;

```

Figure 4.17: Sequence Number Generation in Two Transactions - cont'd

```
/* Second part of transaction begins */

/* Use the sequence number just obtained */
EXEC SQL INSERT INTO Investor
      ( Initials, SeqNo, ... )
VALUES
      ( :Initials, :SeqNo, ... );

if ( sqlcode < 0 )
{
    EXEC SQL ROLLBACK;
    return (1);
}

EXEC SQL COMMIT;
```

Figure 4.18: Sequence Number Generation in Two Transactions - cont'd

Relation "Container"

(Container Number, Block, Stack, Row, Tier,
Other Container Attributes)

Figure 4.19: One Possible Logical Schema for Containers and Their Locations

"gaps" in the sequence. If this is not a problem, then this is a good solution because the transactions become much shorter and concurrency performance improves.

4.2.3 Pre-allocation of records

In a certain container management application at a port, it was required to maintain an inventory of the containers currently in the port's yard [CMC 93]. The yard was divided into large rectangular areas called "blocks". Within each block, containers could be placed on marked locations on the ground, each such location resembling a square on a chess-board. These locations were marked in the X- and Y- directions and were called "stacks" and "rows". Containers could even be placed one on top of the other, and these vertical Z-coordinates were called tiers. So the complete location of a container could be specified by a "cell location" consisting of a block, stack, row and tier.

Now, since each container can occupy only one cell and each cell can accommodate only one container, the relationship between the entities Container and Location is one-to-one. Thus, the logical schema can only be one of two types, as shown in Figures 4.19 and 4.20.

Let us consider the second table "ContainerLocation" in Figure 4.20. This is a unique schema from a data point of view, but it could still be used in one of two ways.

The relation could be empty to start with when there are no containers in the yard. As containers enter the yard, records could be inserted into it. Records could similarly be deleted when containers leave the yard. This is one way to use this relation, as shown in Figure 4.21.

Another way to use it would be to have as many records in the relation as there are "cell locations" in the yard. However, all the Container Number attributes would be NULL

Relation "Container"

(Container Number, Other Container Attributes)

Relation "ContainerLocation"

(Container Number, Block, Stack, Row, Tier)

Figure 4.20: Another Possible Logical Schema for Containers and Their Locations

```

/* Container enters yard */
INSERT INTO ContainerLocation
( ContainerNo, Block, Stack, Row, Tier )
VALUES
( :ContainerNo, :Block, :Stack, :Row, :Tier );

/* Container leaves yard */
DELETE FROM ContainerLocation
WHERE ContainerNo = :ContainerNo;

```

Figure 4.21: An "Insert/Delete" Approach to Relation Usage

```

/* Container enters yard */
UPDATE ContainerLocation
SET    ContainerNo = :ContainerNo
WHERE  Block = :Block
AND    Stack = :Stack
AND    Row   = :Row
AND    Tier  = :Tier;

/* Container leaves yard */
UPDATE ContainerLocation
SET    ContainerNo = NULL
WHERE  Block = :Block
AND    Stack = :Stack
AND    Row   = :Row
AND    Tier  = :Tier;

```

Figure 4.22: An "Update-only" Approach to Relation Usage

when there are no containers in the yard. As containers are brought into the yard, the attributes of the corresponding cells are updated. Similarly, when containers are removed from the yard, the attributes are once again updated and set to NULL. This is illustrated in Figure 4.22.

Which of these methods is preferable?

In this particular application, there were 10,000 cell locations and about 1000 containers in the yard at any given time. Opting for method 2 would mean that 90% of the relation would always be empty. However, with a record width of about a 100 bytes, the entire relation occupied no more than a megabyte, so this waste was not considered high.

It was seen that the application itself tended to cluster the containers in certain well-defined ways. All containers bound for or unloaded from a ship were placed in a separate block. Even within a group of containers bound for a ship, containers meant for discharge at a particular port were always placed in the same stack or adjacent stacks. It was found that a very frequent query was to select all containers belonging to a ship. Another kind of

query was even more detailed. It accessed all containers meant for a ship and destined for a particular port. Thus a clustering of the relation on (Block, Stack, Row, Tier) or (Block, Stack, Tier, Row) was strongly indicated. Even from a concurrency point of view, this clustering turned out to be good because different users were interested in different ships, and therefore in different parts of the yard. Clustering the relation by location separated these users from each other.

This kind of cluster could not be maintained if the records were constantly inserted and deleted, so method 2 was preferred. The relation was therefore static, in a sense. However, there was another frequent kind of query by Container Number ("Where is this container?"). Since a scan of 10,000 records was too high, a secondary index was built on Container Number to expedite this query.

But this led to another problem. *Every change of location for a container* (restacking of containers is frequent in a port) *resulted in changes to the secondary index*, since the Container Number now corresponded to a different tuple id.

In a situation like this, should the designers keep the secondary index (perhaps with frequent reorganisation) or should they live with sequential scan, knowing that with a page size of 2000 bytes and a record width of 100 bytes, the entire table occupies only 500 pages? With a large enough buffer, reading 500 pages should not result in many disk accesses.

To answer such a question, there seems to be no way out but trial and error. In the case presented, the designers found that it was better to keep both the cluster and the secondary indexes. The cost of the scan was more than the index locking and update costs.

4.2.4 Archival

There is a very common situation where "Operations" users access *the most recent records* of a relation for update/insert, while "MIS" users simultaneously access *the entire table* for some ad hoc queries. These two classes of users often conflict because the MIS users may take read locks on the entire table for the duration of their queries. Since it is not possible to predict what kind of queries MIS might make, our earlier technique of storing aggregate information redundantly does not work.

An example of Operations and MIS queries is shown in Figure 4.23.

Keeping the data of all past transactions that MIS might be interested in makes the relation very large. Operations queries then have to search through a very large relation to

Operations

```
/*  
Inserts single record  
*/  
INSERT INTO Ledger  
(...)  
VALUES  
(...);  
  
/*  
Updates recent records  
*/  
UPDATE Ledger  
SET    CheckFld = 'Valid'  
WHERE  Month = :CurMonth;
```

MIS

```
/*  
Scans whole table  
*/  
SELECT AVG( Amount )  
FROM    Ledger  
WHERE   TransType = 'Bills';
```

Figure 4.23: Operations vs. MIS Queries

access just a few records, and this makes them very slow.

If on the other hand, older records not required by Operations are frequently archived and put into another relation, Operations queries no doubt become faster but MIS queries become more cumbersome. MIS users would have to know which records are to be found in which relation.

To overcome this problem, the archival of old records could be done along with the provision of a "view" to MIS defined to be the union of the two relations [CMC 93]. Thus Operations can work on a small and easily manageable relation, while MIS gets a unified picture of old and new data without any additional programming effort. Since MIS queries usually do not update information, a view is acceptable.

4.3 Miscellaneous techniques

In some applications, it is found that different parts of the system become bottlenecks at different times. In one MIS application that we studied [CMC 93], the monthly payroll run necessitated a different set of indexes than normal day-to-day operations. Similarly, the transactions that were run at the end of the financial year required a special set of indexes as well as temporary tables for redundant storage. So it is not always feasible to recommend a *single* physical or logical design schema for optimal performance *at all times*. For example, indexes may need to be dropped before a huge batch load, and then created again.

A really sophisticated design tool should be able to recommend different strategies for different periods of operation. This is different from the problem of determining optimal reorganisation points for databases which has been studied in detail.

Chapter 5

Introducing a Diagrammatic Design Technique

We have incorporated our approach into a diagrammatic method that explicitly captures information on lock conflicts. Many potential bottlenecks show up visually, attracting the attention of the designer and enabling him or her to concentrate on the really crucial areas. The technique is explained in the following sections.

5.1 Extensions to the standard entity-relationship diagram

We assume the reader is familiar with the Chen style of drawing Entity-Relationship Diagrams (hereinafter called ERDs). We make a few deviations in notation from the "classical style", as listed below.

- We show the primary key within the box representing the entity, not as an external attribute, because this is more intuitive when looking at relationships, as will be explained later (See Fig. 5.1).
- We show attributes as small circles, with the attribute names outside them, instead of drawing them as ellipses containing the names. This style is favoured by other researchers as well.
- We do not show relationship cardinalities in the form (min, max) but separately as cardinality ratio and participation, since we believe the latter representation to be more intuitive.

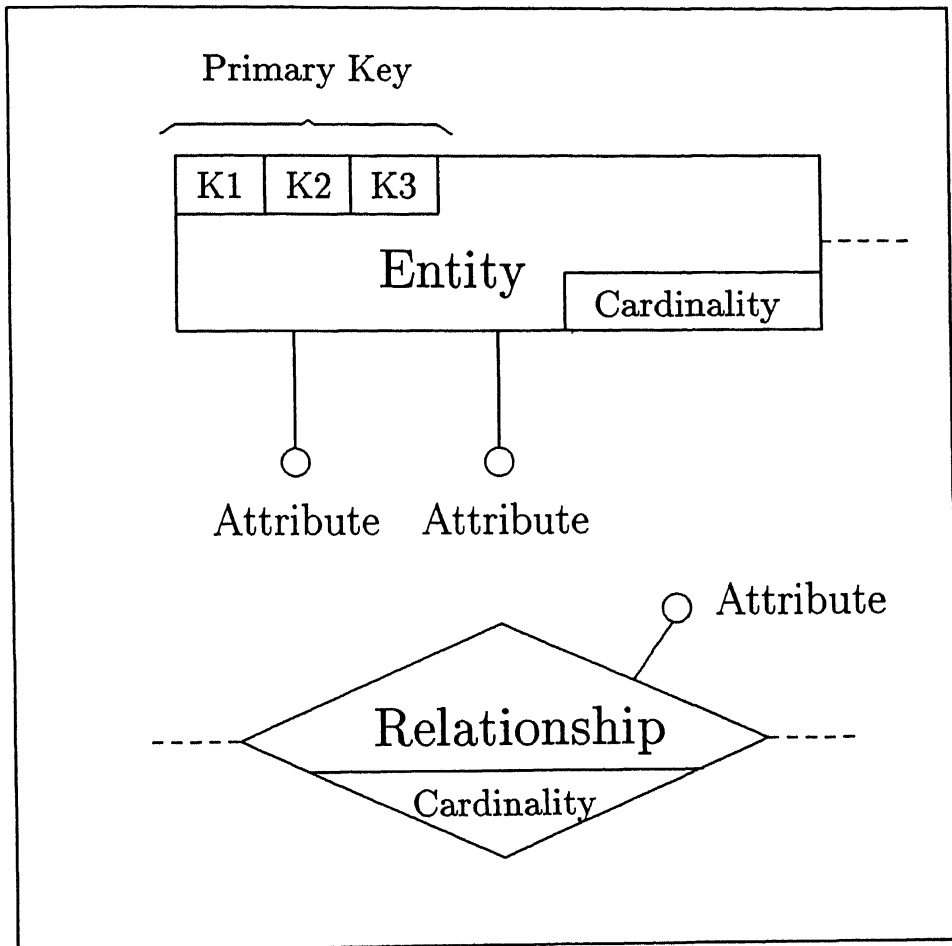


Figure 5.1: Basic Notation - Modified ERD

- We represent total participation by a taut spring, and partial participation by a slack rope, because this more intuitively illustrates the tendency of total participation to lead to a (merged) single relation (See Fig. 5.2).
- We include information on cardinality (the number of instances) along with each entity and relationship. This helps us judge how large a relation is going to be, what proportion of a relation is accessed by a query, whether an index is likely to lead to performance gains, etc. (See Fig. 5.1).

In addition to all the above minor changes to ERDs, we have also attempted to represent transactions in the same diagram, though this is undoubtedly a controversial thing to do.

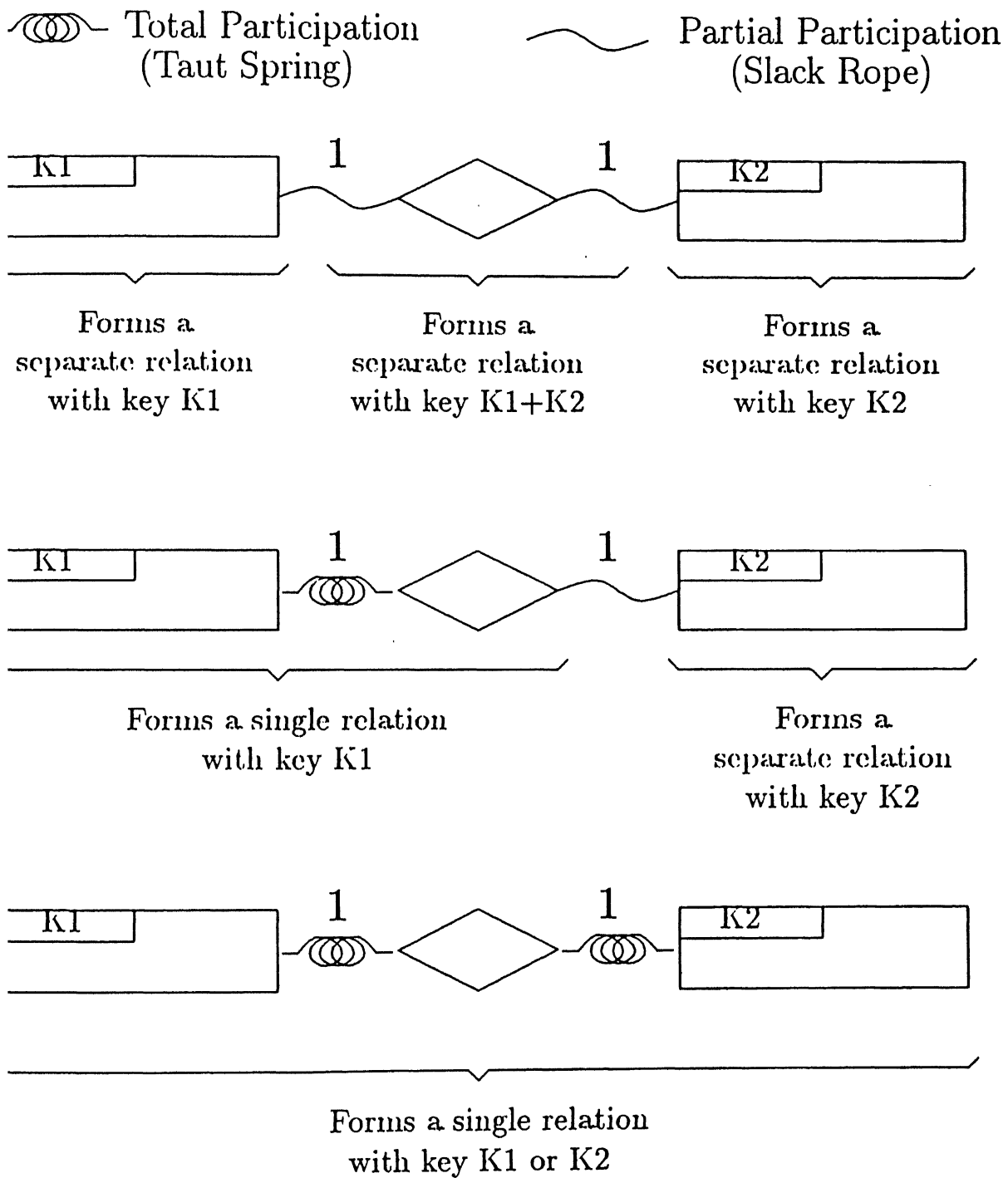


Figure 5.2: The urge to merge, a graphical illustration

In spite of some initial hesitation about incorporating *process* information into what is essentially a *data* representation, we finally decided that the classical ERD was just not using its full modelling potential. After a couple of months of working with the new, extended diagramming method, we find it as natural and intuitive as the original ERD. We could refer to this enhanced notation as ERTD (Entity-Relationship-Transaction Diagram).

There are of course other standard notations available for representing transactions and concurrency, the best-known being Petri Nets [BATI92, PETE77] and Information Control Nets (ICNs) [BATI92, ELLI79]. However, Petri Nets model data *flows*, but not data *stores*. ICNs can represent data stores also, but this representation is typically high level and certainly not of the sophistication found with ERDs. Since our orientation is towards detecting conflicts on data items at the level of relations, pages and even records, we felt it more appropriate to start with a good *data* representation and add transaction information to it, rather than start with a good *transaction* representation and add data information to it.

5.2 Basic notation and methodology of usage

Figures 5.3 to 5.6 provide illustrations of how various types of concepts are represented in this notation. There is a Data side and a Process side to every diagram. The Data side, with the few modifications mentioned earlier, is readily intelligible to anyone familiar with ERDs. The Process side requires some explanation.

Every (operating system) process is represented in the form of a circle or ellipse (Fig. 5.3(a)). In place of processes, we could think of the circles/ellipses as representing user terminals. If there is likely to be more than one such terminal, user or process in the envisaged system, they will be represented as double circles or ellipses. The number of such concurrent processes will also be indicated at some point on the circumference. Double circles or ellipses are visually "heavy" objects and indicate that the transactions linked to them run concurrently, and are likely to contend for locks.

Each of these processes or terminals is capable of firing transactions, which are sequences of individual query statements. The main difference between a transaction and a set of independent query statements is that the transaction holds on to all the locks taken by its constituent statements until it completes. This has major implications for performance,

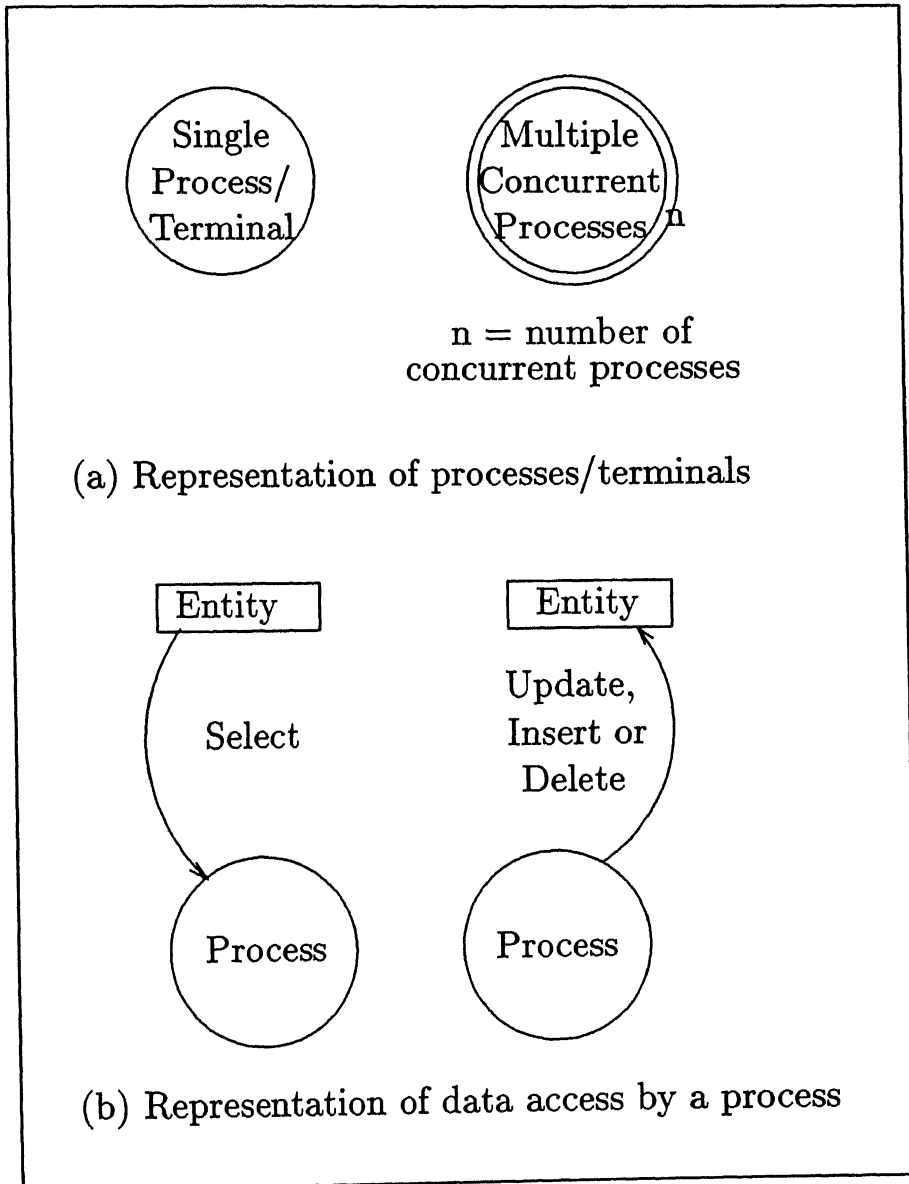


Figure 5.3: Basic Notation cont'd

hence we disagree with the approach of those design tools that merely weight each *individual* query statement by its frequency to arrive at costs of various index configurations.

We can represent transactions explicitly by showing greater detail within each ellipse. We can have several circles inside each ellipse, each representing a stage of processing (See Fig. 5.4). The stages are linked through arrows, showing the sequence of processing. Two independent transactions fired from the same terminal have independent chains of circles inside each ellipse.

Query statements are represented by arrows (Fig. 5.3(b)). Data retrievals (SQL selects) are represented as arrows from the Data side to the Process side. Updates, Deletes and Inserts are shown as arrows from the Process side to the Data side, which intuitively suggests that data is being changed by the statement. One can also learn to see outward arrows from the Data side (selects) as corresponding to shared locks, and inward arrows as exclusive locks (If some entity or relationship has only outward arrows, then it corresponds to a read-only table and will not suffer lock contention). Reading with exclusive locks is shown with a line having arrowheads at both ends (Fig. 5.5).

When *multiple* rows are being retrieved or modified by a query, the arrow corresponding to that query will be a double line, again a visually "heavy" object that draws attention to its importance (Fig. 5.5).

Query statements may also be numbered. A transaction comprising 3 statements may have them numbered 1.1, 1.2, and 1.3. Another independent transaction fired by the same process with (say) 4 statements, would have them numbered 2.1, 2.2, 2.3 and 2.4.

The SQL statement corresponding to each arrow is written near it, using a convenient shorthand. The "where" clause of the statement is particularly important, as will be explained later in this section.

On the Process side, an arrow is connected to the process which makes the query. If detailed transaction steps are shown within each process, then the arrow is connected to the inner circle which represents that stage of processing which makes the query.

It is not difficult to decide where the arrows should be linked on the Data side, either. All the attributes referenced in the query should be found with the entities and/or relationships that the corresponding arrow touches. Note that primary keys of entities are shared by the relationships that connect them, so if a query needs the attributes of a relationship and also the primary key of one of its participating entities, it is not necessary to perform a join

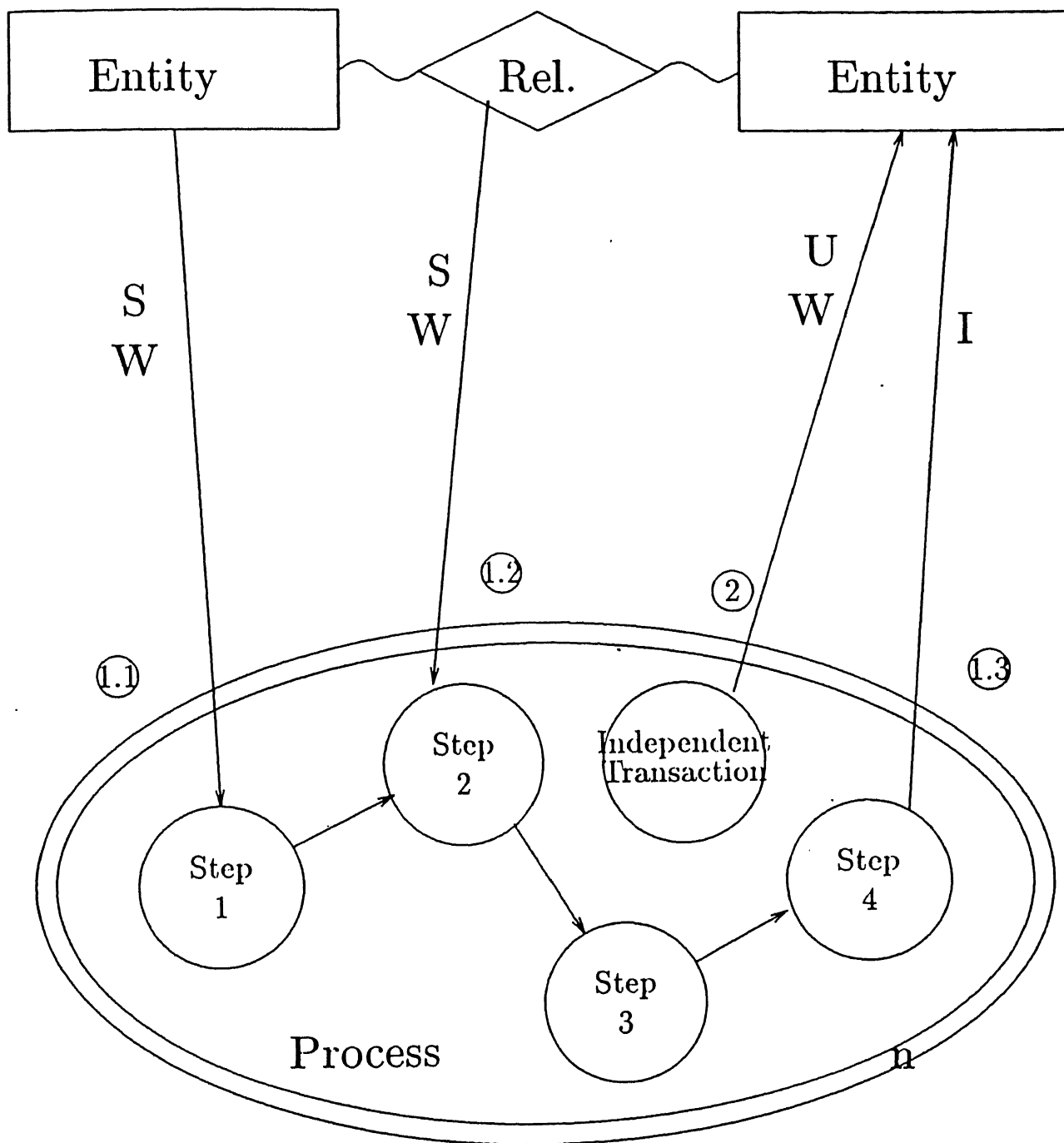


Figure 5.4: Transaction Notation

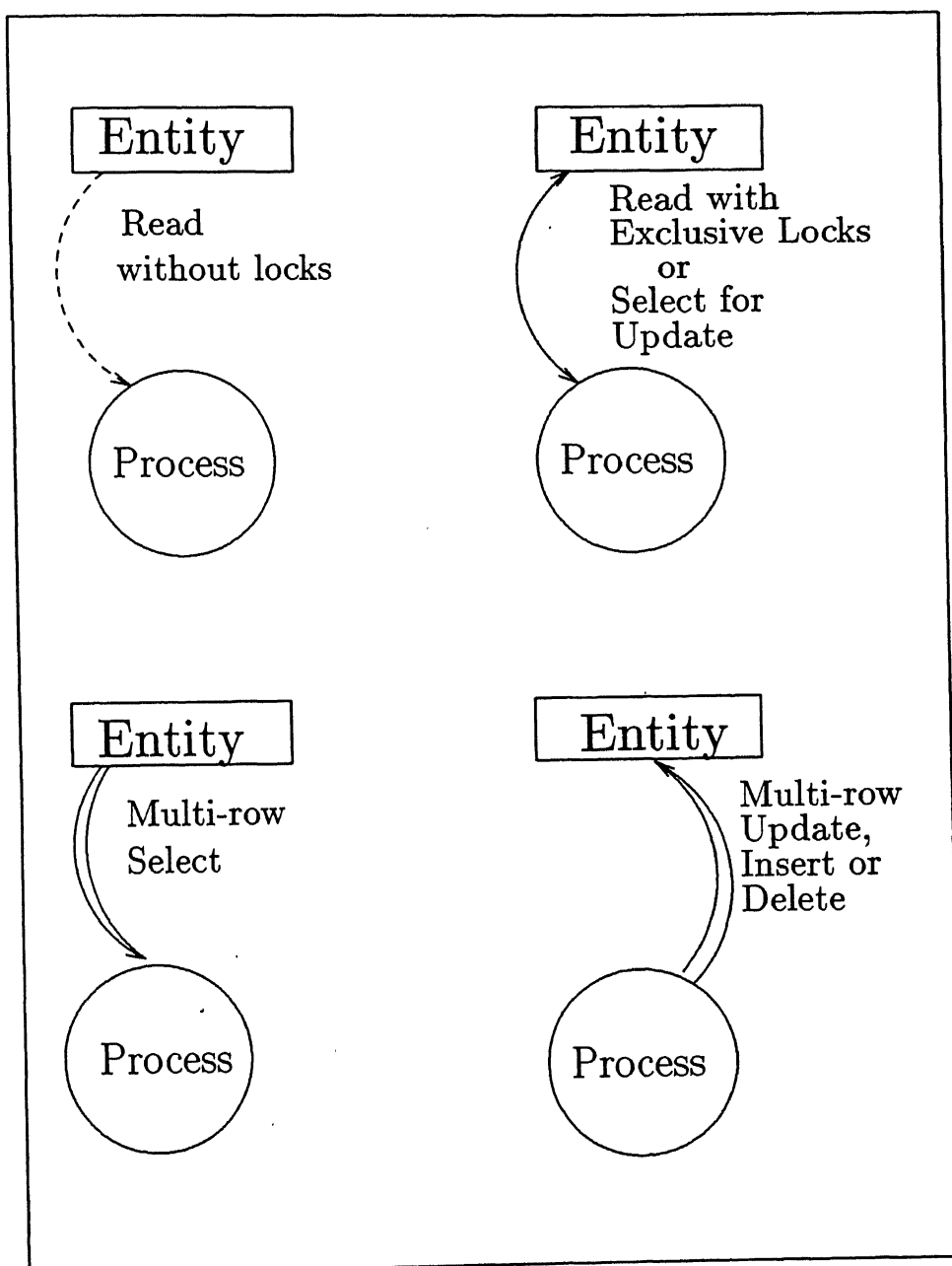


Figure 5.5: Basic Notation cont'd

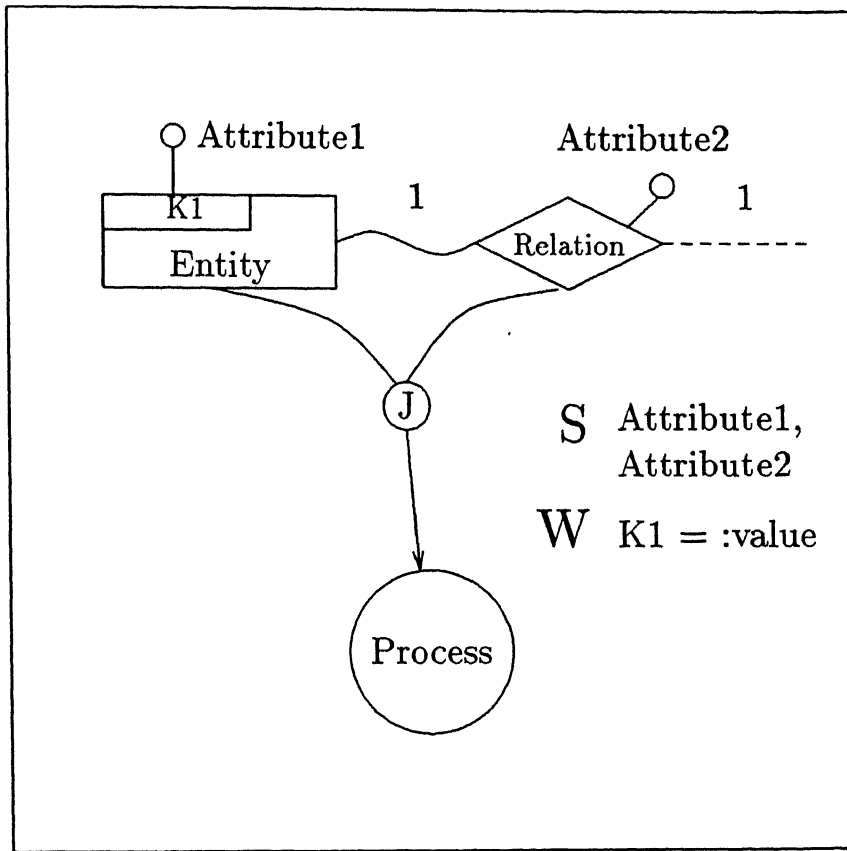


Figure 5.6: A Join

with that entity just to access that key. It suffices to link the arrow with the relationship alone.

If (non-key) attributes from an entity as well as from an associated relationship are required by the same query, then the join that would need to take place between the corresponding relations can be represented by lines starting from the entity and the relationship, joining at a circled 'J' and extending as an arrow to the Process side.

We would prefer not to dwell too long discussing the details of the notation at this stage. It is better to illustrate its use with a few examples, which we will look at in a later chapter. Only two points remain to be made. One is the shorthand for SQL queries, the other a notation for lock conflicts.

5.3 SQL shorthand

We adopt the SQL syntax directly when marking arrows, with some easily-understood abbreviations, as shown in the table. We do not need to specify a relation name, since the arrows show which entities or relationships are being accessed.

<i>Abbreviation</i>	<i>Stands for</i>
S	Select
U	Update
I	Insert
D	Delete
W	Where
A	And
OB	Order by
F	Fetch (next)

5.4 Modelling selection and conflict

We also specify the values taken by attributes, especially in "where" clauses, since this critically affects the set of records selected and thereby the probability of lock conflicts with concurrent transactions.

We *could* write the "where" clause in the form

W attribute = value

However, we need to provide more detail in place of the word "value". We need to say whether the subset of the relation accessed by the query is always the same or variable. If it is variable, does the subset progress "down" the relation with time, or is it random? If there are two queries *of the same type* fired by different users, do they access the same set of records or disjoint sets, or do the sets partially overlap? These situations are modelled using the following keywords:

Subset characteristics:

- W attribute = const (A more-or-less constant subset, such as the set of records in an Employee relation corresponding to employees in a particular department)
- W attribute = var (A variable subset, which could be of the types below)

- W attribute = `chron_var` (A subset that changes with time, such as the latest few records in a relation)
- W attribute = `rand_var` (A subset that changes each time the query is fired, but not in any determinable order, such as in MIS queries)

Conflict characteristics:

- W attribute = `same_const` or `same_var` (The subsets are the same, eg. the queries are interested in exactly the same data records)
- W attribute = `diff_const` or `diff_var` (The subsets are disjoint, eg. departmental managers query on the records pertaining to their own departments)
- W attribute = `overlap_const` or `overlap_var` (The subsets overlap partially or, in page-locking systems, the records selected by two queries are likely to lie on the same page)

It should be noted that the above keywords denoting conflict are only for queries *of the same type* fired by different users or terminals. This notation cannot be used to model the conflicts that occur between two different kinds of queries.

It requires some judgement to decide which of these words to use, but the rewards are worth the effort. We now consider some of the important heuristics that experienced designers use.

- W attribute = `chron_var` (A subset that changes with time, such as the latest few records in a relation)
- W attribute = `rand_var` (A subset that changes each time the query is fired, but not in any determinable order, such as in MIS queries)

Conflict characteristics:

- W attribute = `same_const` or `same_var` (The subsets are the same, eg. the queries are interested in exactly the same data records)
- W attribute = `diff_const` or `diff_var` (The subsets are disjoint, eg. departmental managers query on the records pertaining to their own departments)
- W attribute = `overlap_const` or `overlap_var` (The subsets overlap partially or, in page-locking systems, the records selected by two queries are likely to lie on the same page)

It should be noted that the above keywords denoting conflict are only for queries *of the same type* fired by different users or terminals. This notation cannot be used to model the conflicts that occur between two different kinds of queries.

It requires some judgement to decide which of these words to use, but the rewards are worth the effort. We now consider some of the important heuristics that experienced designers use.

Chapter 6

Heuristics

Many heuristics exist for converting Entity-Relationship Diagrams into relations in Third Normal Form, and these are fairly standard and well-known [ELMA89]. We restate these with some minor modifications and add our own performance-related heuristics.

1. **ENTITY TRANSLATION RULE** - In general, translate every entity into a relation. A weak entity will translate into a relation, too, but it will have the primary key of its parent entity tagged on to its own primary key to form the complete key of the relation.
2. **RELATIONSHIP TRANSLATION RULE** - In general, translate every relationship also into a relation. The primary keys of all the participating entities will be linked together to form the primary key of the resulting relation.
3. **RELATION MERGING RULE 1** - If there is a 1-to-1 relationship between two entities, then translate into a single relation wherever there is a spring denoting total participation (See Fig. 5.2). If the spring is only between one entity and the relationship, then the resulting relation will contain all the attributes of the totally participating entity, the primary key of the other entity and the attributes of the relationship. The other entity which does not participate fully will form its own relation. If both entities participate totally (two springs), then only one relation is needed in all. Merging is to be avoided where a slack rope exists, since there will be null values in the resulting relation.

4. RELATION MERGING RULE 2 - If there is a 1-to-N relationship and there is a spring on the N side, merge the entity on the N side and the relationship into a single relation. If there is a spring on the 1 side, ignore it.
5. SECONDARY INDEX RULE - If a *single* row is being selected from a relation based on a "where" clause, then a secondary index on that attribute may be called for. In such cases, it is enough if the system is able to *determine* the location of the record quickly. There is no gain involved in *sorting* the relation using a primary or cluster index (This is all the more important when unrelated updates, inserts or deletes are also taking place, because in the absence of an index, the RDBMS might lock the whole relation in order to provide Repeatable Read (RR) and guard against the phantom tuple problem, which is discussed in [KORT91]. The presence of the index will result in less conservative locking and improve concurrency).
6. PRIMARY/CLUSTER INDEX RULE - If a *group* of records is being selected based on a "where" clause, you might need a primary/cluster index on those attributes. This is because it is not sufficient to merely determine where the desired records are located but to have them *physically together*. If they are located on different disk cylinders, I/O will take a long time, increasing the duration of the transaction. Hence group selects should ideally find all their records in one place, in as few (contiguous) disk blocks as possible. Even if it is not possible to guarantee physical contiguity of logically contiguous disk blocks due to the way the Operating System assigns blocks (as in Unix), contiguity of records should be assured at least within a block. This will minimise disk I/O.
7. FULL SCAN RULE - If ALL the records of a relation are being selected, then no index (primary or secondary) may be required, except if there is an "order by" clause, in which case a primary/cluster index is required on that attribute. An index does not really speed up matters when the entire relation is being accessed because all the records in each block will be read anyway, so there are no wasteful reads. Clustering helps to avoid sorting when "order by" is used, because the records are already found in that order in each block.
8. INDEX AVOIDANCE RULE - If there is a select based on a certain attribute, then an index is indicated as mentioned above, provided no inserts or deletes take place on

the relation, also no updates to the indexed attribute. If such changes take place, and they are frequent compared to the select, then the index is not recommended. This is because such changes to the relation change the index as well. Indexes also have to be locked when being modified, and this is usually more restrictive than locking base relations. Contention between transactions is worse when index updates are involved, hence such situations should be avoided unless the benefit of the index is likely to be more than its cost. See [SRIN91].

9. HORIZONTAL PARTITIONING RULE - If a relation is being used predominantly for inserts, especially in a page-level locking system, then the table may have to be horizontally partitioned based on some attribute value, or by some round-robin or hashed sequence, otherwise the inserts will end up being strictly serialised, resulting in poor performance. This is because inserts to a relation are typically appends to its last page. Since an exclusive lock is required for this operation, only one insert will be allowed at a time. This is far too slow when transactions are large, or the number of concurrent users is large, or both. Horizontal partitioning in effect creates many relations, each with its own queue of pending transactions. Thus a single queue gets broken up into a number of queues, reducing lock contention.
10. VERTICAL PARTITIONING RULE - If different attributes are accessed by different transactions, vertical partitioning is suggested. Each such set of attributes should be placed in a separate relation with the primary key of the original relation repeated in each. This is because access is needed only to attributes, but locks have to be taken on records or pages. To illustrate, even if two transactions attempt to access *independent* attributes, they would be in conflict if they accessed the *same record or page*. Vertical partitioning tries to place such independently accessed attributes in separate relations, so that such unnecessary locking does not take place.
11. AGGREGATION RULE - If aggregation queries are encountered, redundant storage of the aggregate value in another relation is suggested [CMC 93, PEIN88]. The aggregate function could be any of MIN, MAX, AVG, COUNT or SUM (Most queries that require the *latest* of several values also translate into an aggregation because they perform the MAX operation on a date or timestamp). If the aggregation query has a "where" clause, then the new relation which stores the aggregate value will have

those attributes as the primary key. If the aggregation is on the entire relation with no "where" clause, then the aggregate value is the only row and the only column of a new relation.

12. **DISJOINT SET SELECTION RULE** - If disjoint sets of records are accessed by different transactions (i.e. there are no records in common between the sets accessed by these transactions), clustering of the relation is recommended on the attributes that distinguish these sets from each other. Horizontal partitioning on the basis of these distinguishing attributes may also be considered.
13. **HASHING/DECLUSTERING RULE** - If different transactions access single records that lie on the same page, hashing/de-clustering based on the attributes referenced in the transactions' "where" clause may be explored.
14. **JOIN AVOIDANCE RULE** - If there is a frequent join between two objects, consider merging into a single relation so that the join becomes a simple select.
15. **READ WITHOUT LOCKS RULE** - If there are transactions which read a record and update it based on its value, and if the update rarely takes place, then it is advisable to first read without locks and then if the update seems necessary, to redo the operation with locks [PEIN88].
16. **TRANSACTION SPLITTING RULE** - If all transactions that access a set of relations are of the same type (i.e. the same program is run at different terminals by different users) and if each such transaction is very long, and some prefix of the transaction (i.e. the first few queries) is idempotent (i.e. can be run repeatedly without causing inconsistency even when interleaved with other transactions), then the transaction may be split into two at this point.
17. **PRE-ALLOCATION RULE** - If there are frequent inserts and deletes of records in a relation, but clustering on some attribute is required by another transaction, consider allocating all required records initially and merely updating values instead of inserting and deleting records.
18. **ARCHIVAL RULE** - If some transactions access a small (latter) part of a relation but other (reader) transactions access the entire relation or large parts of it, consider

archiving older records into another relation and letting the readers view the *union* of the two relations. This gives writers a small and manageable set of records while keeping the queries of readers simple.

Chapter 7

Case Studies Revisited

In a previous chapter, we had presented several cases of performance problems and shown how each was actually solved by human designers. In each case, the basic problem was one of lock contention. Some of the solutions followed directly from a proper understanding of the problem, other solutions were more innovative and called for some creativity on the part of the designer. Our aim has been to develop a technique that points a designer in the direction of such solutions. In the following sections, we revisit each of those problem cases and show how our proposed diagrammatic technique (along with the heuristics listed before) helps us arrive systematically at the same solutions.

When searching for locking problems, we visually scan the Data side of our diagrams and examine each entity and relationship in turn. Since each of these becomes a relation in the general case, heavy access on any of these suggests possible lock conflicts. Heavy access could either be in the form of multiple arrows converging on it, or in the form of an arrow from a double circle, or both.

7.1 Horizontal partitioning

In many applications, there are concurrent users who wish to insert records into a single relation. One example we studied was that of stockbrokers at a computerised stock exchange inserting bid records into a BidsFor relation (See Fig. 7.1).

In such cases, there is heavy contention for the last page in page-level locking systems. This is illustrated by the arrow marked 'I' going from a double circle into an entity.

It is obvious that multiple users are inserting records one by one into a single relation

(since an entity gets converted into a relation). The Horizontal Partitioning Rule tells us that the relation has to be partitioned on one of its attributes, but it doesn't tell us which one.

At this stage, we could decide on any attribute for this partitioning. We could also decide to use a round-robin or hashed partitioning scheme, if the DBMS provides it, but we observe that there is a second set of processes that access groups of records from the BidsFor relationship based on the ScripCode attribute. Since a group of records is being selected with ScripCode being the attribute in the "where" clause, the Primary/Cluster Index Rule tells us to group them by ScripCode. Also, since the sets being selected are disjoint ($\text{ScripCode} = \text{diff_const}$), the Disjoint Set Selection Rule tells us to either cluster or horizontally partition the table based on this attribute. So we are led directly to the solution of horizontal partitioning based on the ScripCode attribute, which in this case is what human designers actually did.

(We need to incorporate information on frequencies and volumes into the diagram because a similar system with a very low transaction rate (low relative to the power of the computer) may not require such partitioning. Clearly, there is scope for further formalising the technique by factoring in the power of the computer to be used, and thereby deriving whether partitioning is required at all, and to what degree.)

7.2 Vertical partitioning

We also consider the case of different types of users making queries on the same relation but accessing different attributes of that relation. This can be seen illustrated in Fig. 7.2. This is an interesting problem. In this particular case, we can see that the two high-frequency queries are both selects, so partitioning may not be required at all, since there is likely to be very little contention.

If some of the updates are also frequent, the level of contention will be higher, and it may be required to split the table, in which case there are three candidate relations. All three will have EmpNo as the primary key. The first will be clustered by Grade to aid the query of Personnel, and will also have the Salary attribute. The second will be clustered on Project to speed up the Project Managers' queries. The third will be clustered on Department for the benefit of the Department Managers' queries. The attributes Name and Grade feature

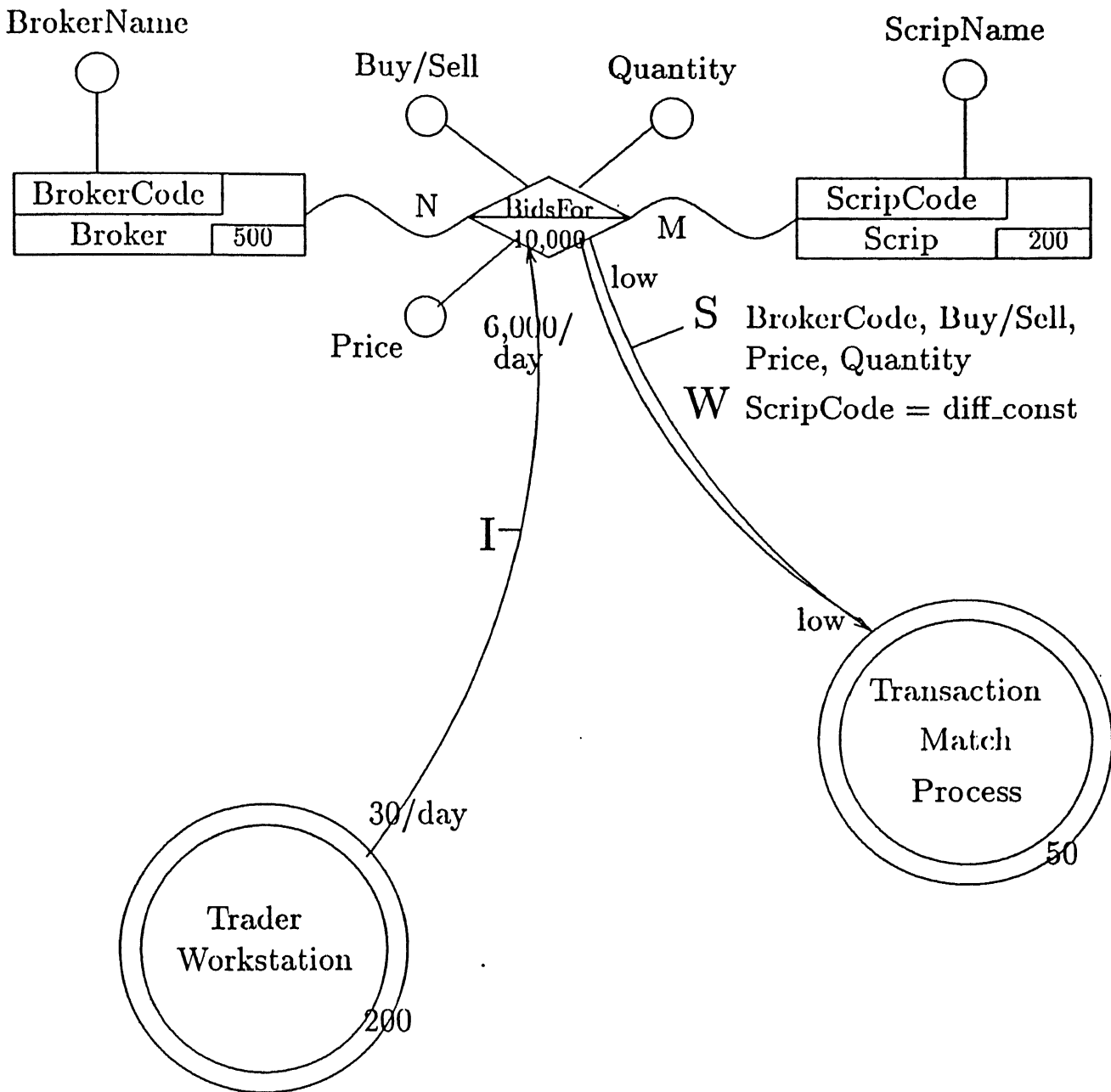


Figure 7.1: High Insert Load and Subsequent Contention for the Last Page

in the queries of more than one type of user, so it might be necessary to either store these attributes redundantly in more than one of these relations, or to perform a join between two relations. Looking at the query frequencies, it should be possible to make these decisions. The attribute Address may be stored in any of these three relations.

On the other hand, a design technique that divides attributes into two groups (frequently accessed and infrequently accessed) would recommend only two relations with attributes as shown:

(EmpNo, Name, Grade, Salary, Project, Department)

(EmpNo, Address)

Clearly, this does not solve the contention problem, as explained in an earlier section. In fact, it makes the problem worse.

7.3 Redundancy

We had earlier seen a case where investors at a computerised stock exchange were being registered by the assignment of running serial numbers appended to their initials. Each unique set of initials had its own sequence of numbers going from 0000001 onwards. The application program registering a new investor had to first check the database for the last serial number assigned to an investor with the same initials, add 1 to it, then compose the unique, identifying code for this investor by appending this new number to his initials. As can be seen from Fig. 7.3, there are about 10 concurrent users selecting records from the relation and inserting new ones. (Though the select returns a single, aggregate-valued record, it locks the entire set of records which match the initials in its "where" clause, in order to ensure Repeatable Read. Hence a double arrow is shown).

Inspection of this diagram shows up some problems immediately. The select accesses and locks a large number of records. We know that a cluster index is probably needed to speed up such group selects, (Primary/Cluster Index Rule) but then we also know that indexes make performance worse when there are inserts (Index Avoidance Rule). The contradiction makes us realise that this operation is likely to be a slow one in any case, unless the basic schema or transaction logic is changed.

Another important aspect also becomes obvious on closer inspection. Looking at the two arrows that connect the Data side and the Process side in the diagram, we see that

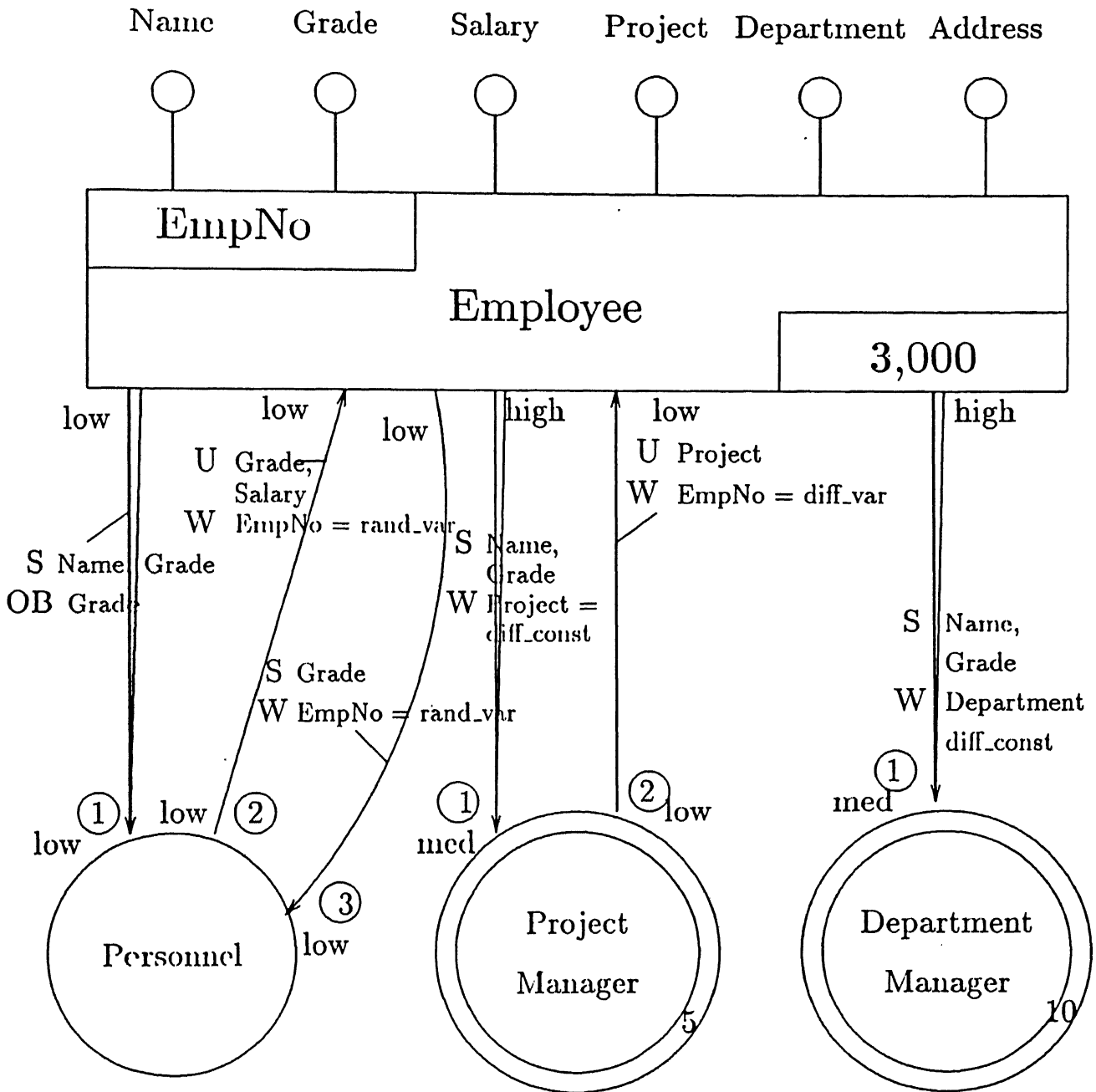


Figure 7.2: Vertical Partitioning based on Attributes Accessed

they form a cycle. Whenever multiple processes (as represented by double circles or ellipses) are involved in such cycles, there is the possibility of deadlock. As we pointed out earlier, select arrows can be looked at as shared locks, while update, insert and delete arrows are exclusive locks. In the diagram, we see that multiple processes may concurrently succeed in reading the same records from the table, since the select involves no conflict, but when the processes try to update, a deadlock will result because no process can acquire an exclusive lock when another process has a shared lock.

We now have to change the schema or the transaction (or both) in order to speed up the process and avoid deadlocks. As a first step, whenever there is a contradiction between the need to have a cluster index and the need to avoid it, we should consider redundancy instead of aggregation. The "where" clause of the group select has only the attribute Initial, but the entity Investor has attributes Initial as well as SerialNo. So to move towards redundant storage of something that depends only on Initial, we should separate out that part of Investor that deals with Initial.

We modify the diagram by introducing a new entity called Initial as shown in Fig. 7.4. Investor now becomes a weak entity dependent on Initial. There is a 1-to-N relationship between Initial and Investor, and this is now the one accessed by the processes.

Now the group select with Initial in its "where" clause can be converted into a single-row select on the entity Initial (see Fig 7.5). The aggregation MAX(SerialNo) can be converted into a simple attribute LastSerialNo attached to the entity Initial. The transaction accordingly gets modified, too. Since redundant storage adds extra steps to a transaction to keep the derived value up to date and consistent, two extra (alternative) steps are added as shown (1.2a and 1.2b). Now the relationship "Is the Initial of", when translated into a relation, will have only inserts. The standard techniques for dealing with insert-only relations can be used here. The entity Initial will be translated into a relation which has single-row selects and single-row updates based on the same "where" clause, and also some inserts. By the Secondary Index Rule, we are led to the decision to build a secondary index on the attribute Initial of this relation. The Index Avoidance Rule tells us that such an index is to be avoided if the indexed attribute is frequently updated (which is not so in this case), or if there are frequent inserts or deletes. Here, there are inserts but they are not frequent compared to the other operations, hence the index may be retained.

Deadlock can be avoided by reading with exclusive locks. The use of a double-headed

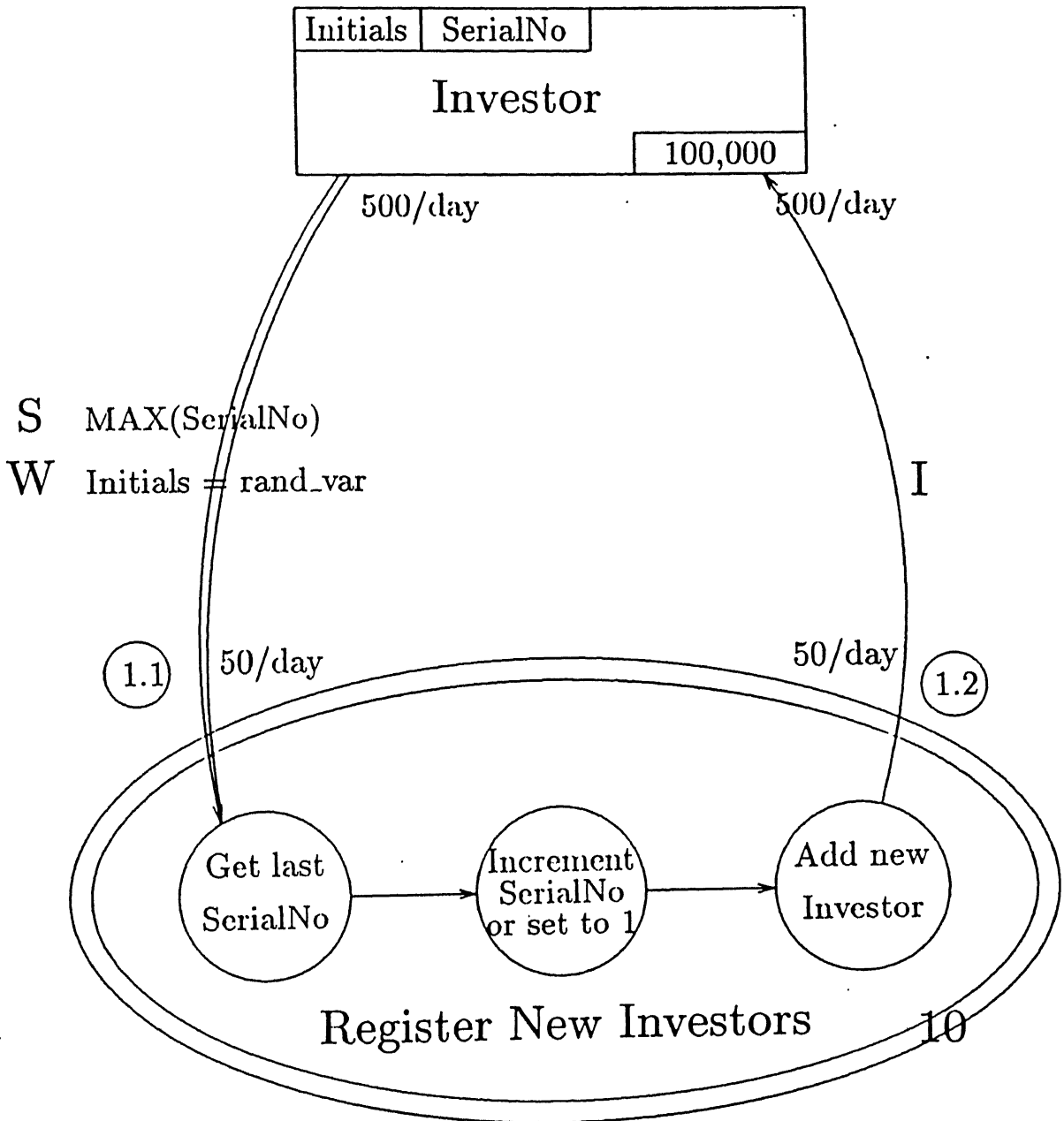


Figure 7.3: Sequence Number Generation - a Naive Approach

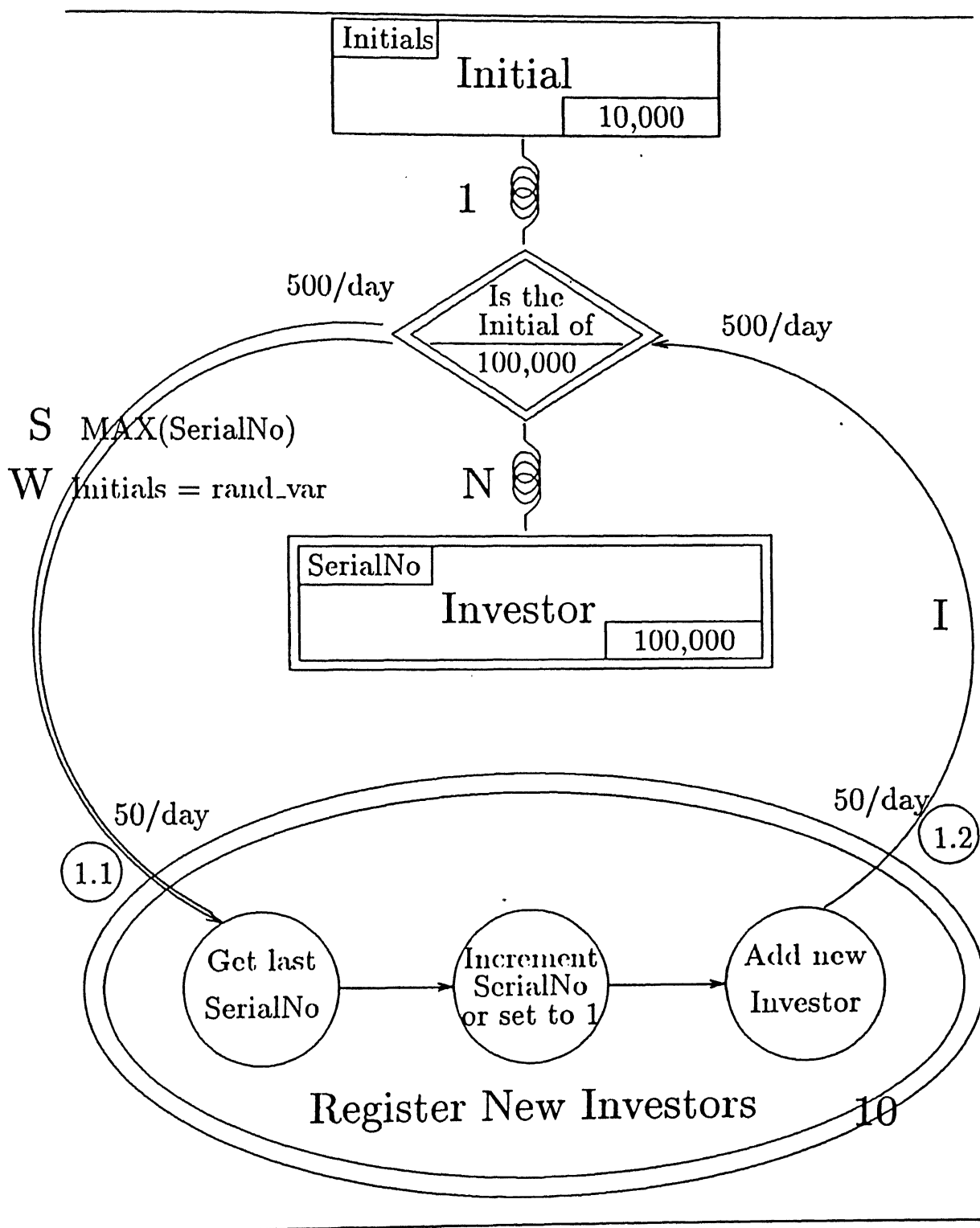


Figure 7.4: Sequence Number Generation - Towards a More Efficient Solution

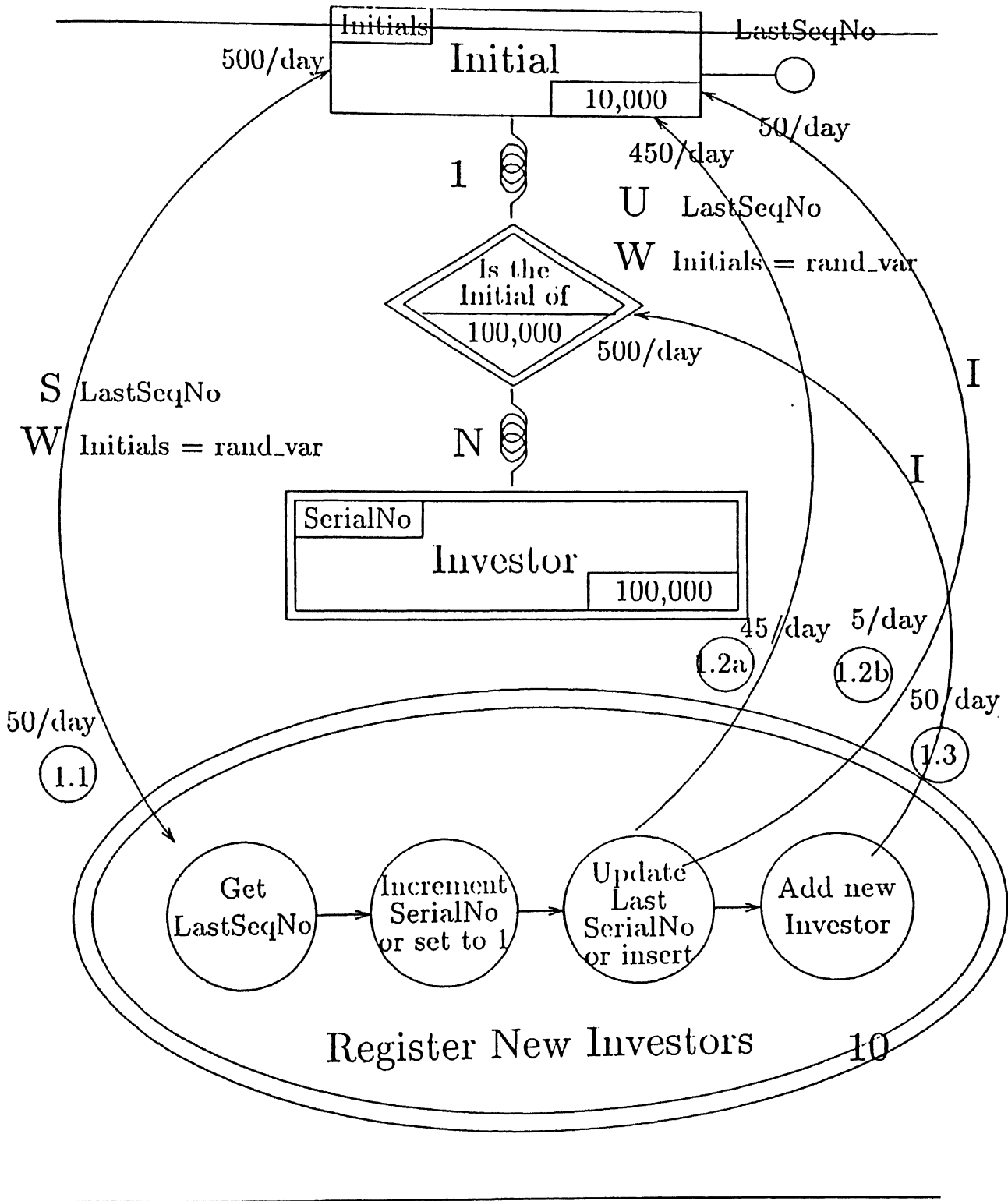


Figure 7.5: Sequence Number Generation - Redundant Storage

arrow visually shows that there is no cycle now. Such exclusive locking need not cause concurrency problems, either. In row-level locking systems, there will be very little conflict, since each query accesses a single record and not a large set of records as in the original situation. Even in page-level locking systems, a lower fill factor for data pages will result in fewer unwanted records being locked.

Thus, the diagrammatic method, in conjunction with heuristics, can systematically lead to the solution which was actually found by human designers through trial and error.

7.4 Clustering

Fig. 7.6 shows an employee database which would probably be translated into three relations, Employee, Department and WorksFor. The major users are departmental managers who fire two types of transactions, as shown in the figure. Both the transactions (which are single statement transactions) involve joins. The Join Avoidance Rule and Entity Translation Rule 2 tell us to merge WorksFor and Employee. The join is thus avoided, but the basic problem of the group select remains, as also the possibility of page conflicts between the transactions in page-level locking systems. Since multiple records are being selected and since the "where" clause has a diff_const in it, the Primary/Cluster Index Rule as well as the Disjoint Set Selection Rule tell us to cluster the merged relation on the attribute DeptCode. The update has an additional "where" clause that selects records based on date, but since the statements are linked through an AND, this does not affect the major decision of clustering on DeptCode.

(Though it appears that a deadlock cycle exists, it is not so because the two arrows do not belong to the same transaction. In fact, if more detailed circles representing stages of processing were shown within the double circle, it would be seen that the two arrows were attached to two different inner circles. The cycle would not be complete on the Process side.)

7.5 Reading without locks

[PEIN88] describe a situation at a stock exchange where many concurrent processes perform the same sequence of operations - check to see if the currently quoted buy (sell) price is greater than (less than) the greatest (least) price in the database so far, and if so, update

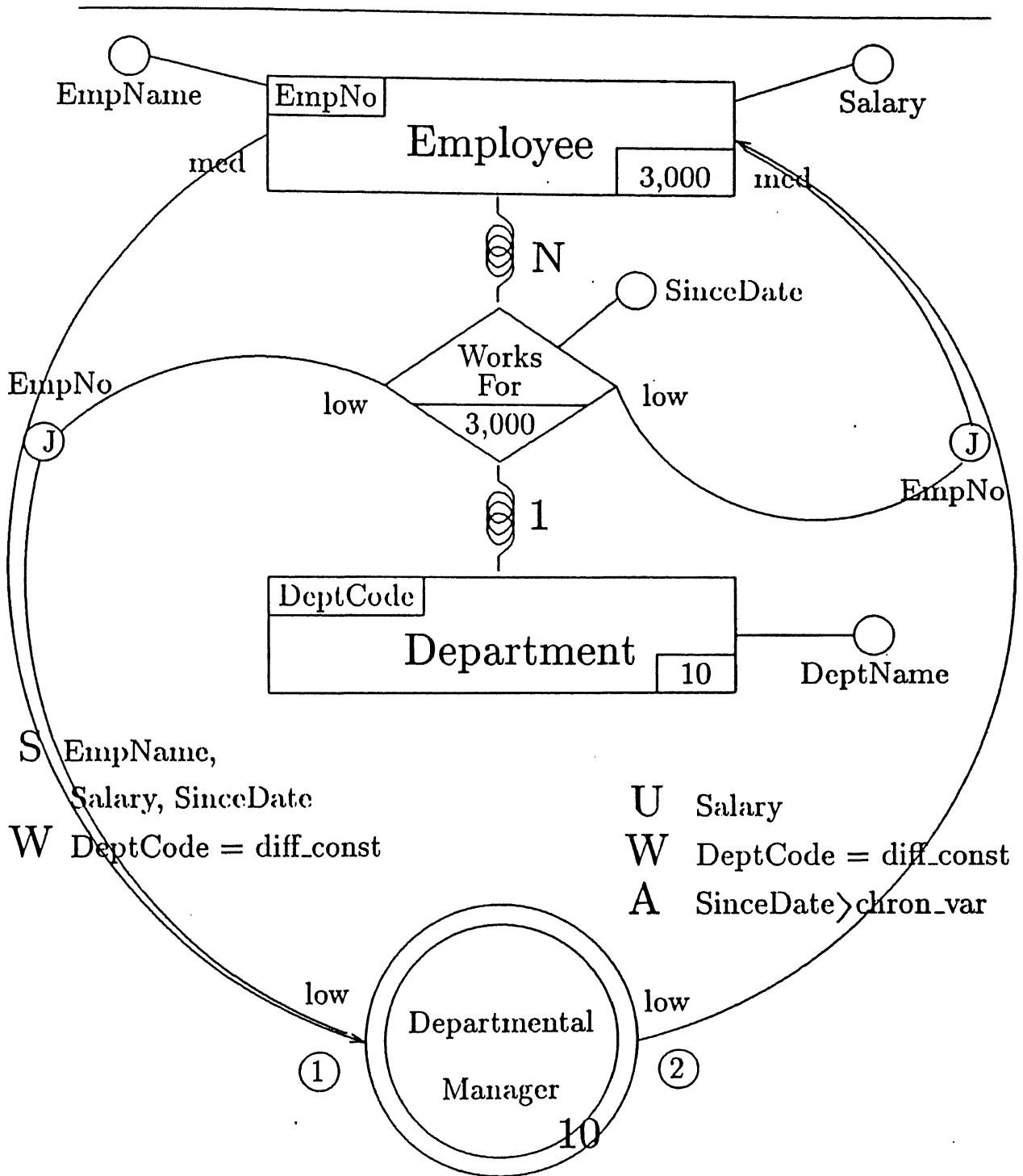


Figure 7.6: The Case for Clustering

the value in the database. It was found that most of the transactions took read locks unnecessarily because their currently quoted buy (sell) prices were usually never greater (less) than the value stored in the database, so the update was rarely necessary. This frequent locking of the relation in shared mode hindered the few transactions that really had to update the relation, so the designers modified the transactions to first read without locks. If it turned out that the update was necessary, the entire transaction would be redone with locks. Though this seems wasteful, the low frequency of the update makes the benefit of reduced locking worth the cost of redoing some transactions.

The original situation is represented in Fig. 7.7. As before, we can see the cycle denoting potential deadlocks. One way to break the deadlock cycle is to read with exclusive locks, but this strictly serialises the transactions. To find a better way, we examine the diagram more closely to find the path of greatest "flow". We see that a very minor part of the transaction actually participates in the second part of the transaction. Our aim is to reduce the frequency of the deadlock cycle, if not eliminate it altogether, so we would like to move the read operation further down the chain of processing stages so that it comes after the "fork" which determines whether an update is required or not. But since the "fork" depends upon the result of the read, this looks impossible at first.

Fig. 7.8 however shows how this may be done. The dashed arrow represents a read without locks. This does not contribute to the cycle. After the "fork", another read is performed, and the transaction goes through as before. The frequency of the cycle is drastically reduced compared to the first case. Now it is even feasible to read with exclusive locks and break the deadlock cycle. This is a change to the transaction design as opposed to schema design which we were looking at till now. This called for some innovation, since the solution does not directly flow from the diagram. As this situation is likely to be encountered frequently, we have stated the solution in the form of the Read Without Locks Rule.

7.6 Splitting transactions

As we discussed earlier, we must try to split transactions into smaller ones wherever possible. Consider the solution to the sequence number generation problem discussed earlier (Fig. 7.5). This no doubt marked an improvement over the original situation of Fig. 7.3, but

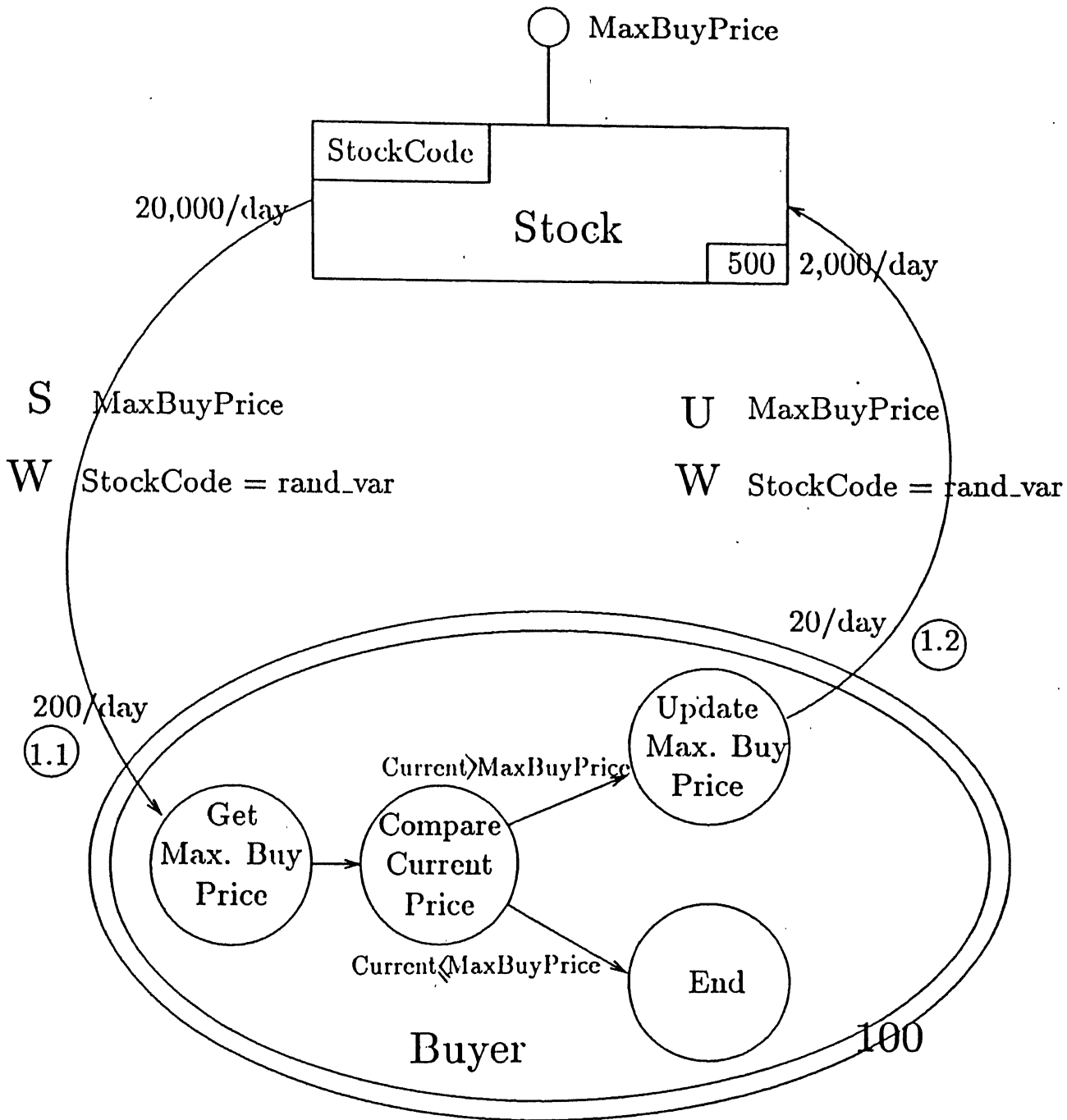


Figure 7.7: Stock Trading with High Locking Contention

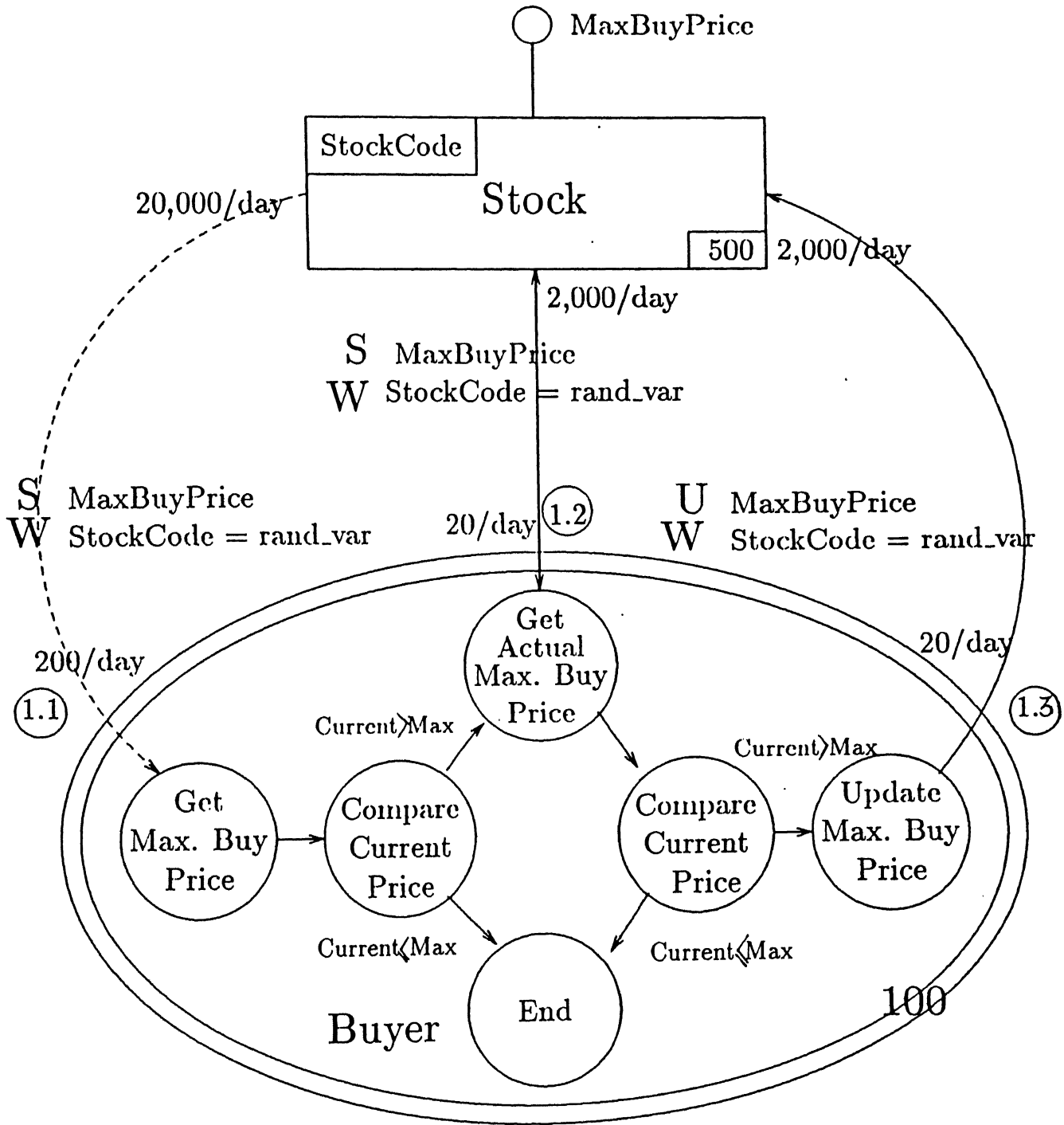


Figure 7.8: Reading without Locks to reduce Locking Contention

the transaction is still too long (4 steps of which 3 are database accesses). We know that long transactions contribute heavily to lock conflict [GRAY78], so we want to split up this transaction into smaller ones.

We examine the diagram and see that one entity and one relationship are being updated by the transaction (Even the relationship "Is the Initial of" is likely to be merged with the entity Investor by Relationship Translation Rule 2). We try to reorder the processing stages so that all accesses to one of them are completed first. In this case, we find that the existing order itself is satisfactory. The entity Initial is used first and disposed of. The relationship "Is the Initial of" is accessed only after that. So it *may* be possible to split the transaction at this point. To check this, see whether the first part of the transaction is idempotent with respect to the application, i.e. can it be run repeatedly without causing inconsistency?

In this case, we see that it is possible. The split transaction is represented in Fig. 7.9, the dashed line between processing stages symbolising the end of one subtransaction and the start of another. This is a change to transaction design which cannot always be derived mechanically, so we have added this to our rule base as the Transaction Splitting Rule.

7.7 Pre-allocation of records

We saw the example of a container management system at the container terminal of a port. Containers enter and leave the port's "yard", and this is kept track of by the computer. The location of a container in the yard is called a "cell", and this is specified by three dimensions - stack, row and tier. Each such group of cells arranged in a cuboid is called a block, and there are several such blocks in the yard. Hence the complete address of a container is given by a block, stack, row and tier. Most cells in the yard are empty. Only about 10 - 25 % of the cells are likely to have containers in them (say). The major port operations affect this database as shown in Fig. 7.10. We see that several operations access the container location based on the block and stack, so the Primary/Cluster Index Rule dictates that a cluster index be built on block and stack. However, the Index Avoidance Rule points to the opposite conclusion because of heavy inserts and deletes. Whenever we have such a contradiction, we usually consider redundancy. Here, we can consider another alternative as well.

We can pre-allocate records corresponding to all possible cell locations, even if no con-

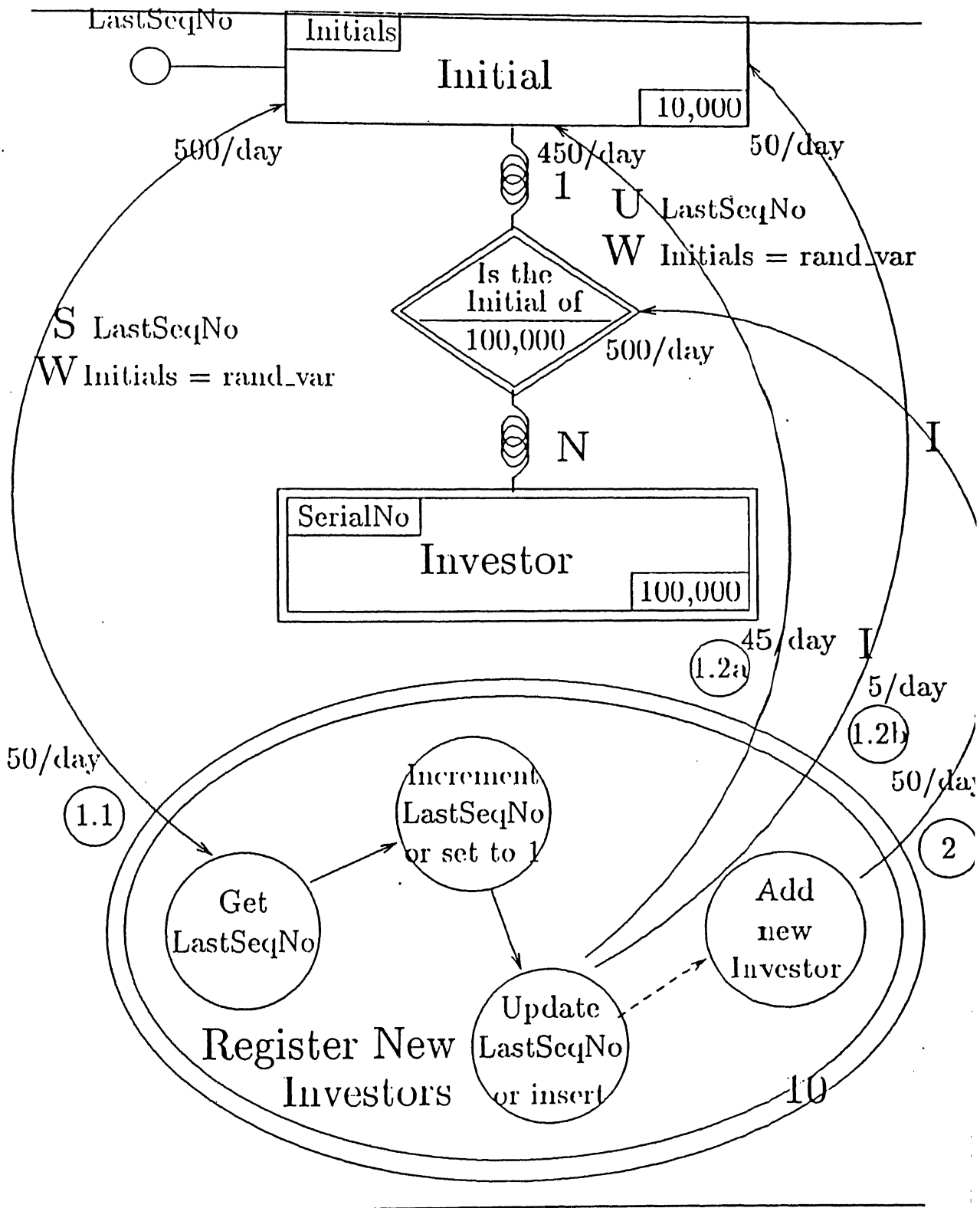


Figure 7.9: Sequence Number Generation - Transaction Splitting

tainers are in them. Whenever a container enters the yard, instead of inserting a record, we can merely update the corresponding cell's record. Similarly, whenever a container leaves the yard, instead of deleting the record, we can just update the record, setting the container number to null. This enables us to use the Primary/Cluster Index Rule without contradicting the Index Avoidance Rule. What we lose is storage efficiency. This new approach occupies more space due to pre-allocation of even those records that are not required. If this is OK, then we have managed to speed up our operations. This is illustrated in Fig. 7.11.

7.8 Archival

We looked at the situation where "operations" users access the latest records in the database while "MIS" users may potentially access any records, usually large groups of them at a time. Fig. 7.12 illustrates such a situation. The MIS users with their conversational transactions and large aggregation queries can hold read locks on tables for long periods, holding up operations users. In addition, because of the requirement of MIS users, older records which may be useless to operations may have to be retained in the database. This slows down operations queries which now have to search through large relations to find relatively few records. We can see from the diagram that the operations queries access the most recent records only. Archival of older records into other relations solves this problem but makes it difficult for MIS users to query on both these sets of relations. What we call the Archival Rule tells us to provide a view to MIS of a union of both these sets of relations, production as well as archive. This will result in operations queries being expedited, and at the same time, will not make MIS queries cumbersome.

It appears as if the diagramming technique may not help in *deriving* the necessary changes to transaction design as automatically as it led to schema design decisions. However, it highlighted the performance bottleneck areas in all the above four cases, and the designer was then able to concentrate on easing these bottlenecks.

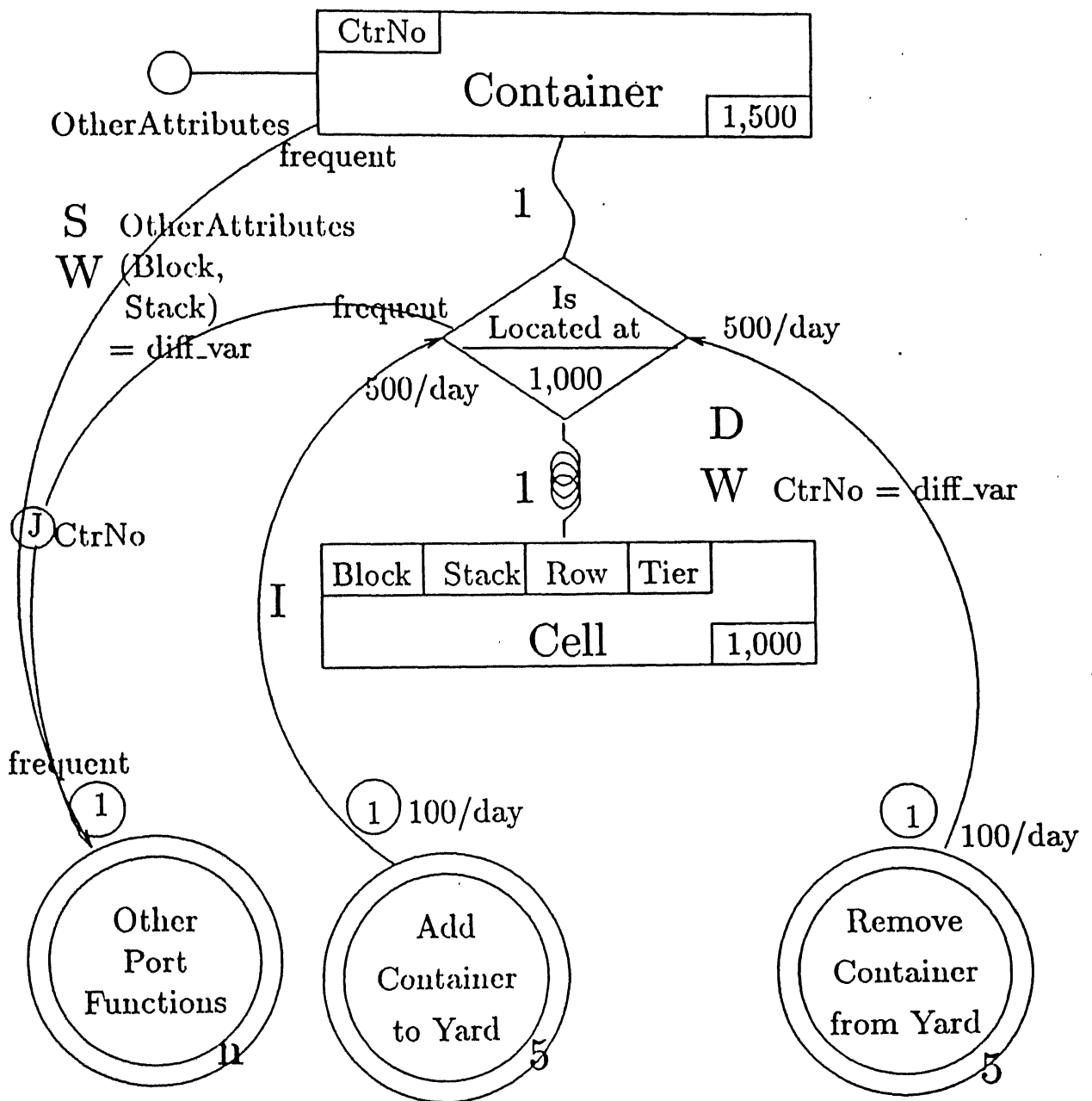


Figure 7.10: An "Insert/Delete" Approach to Relation Usage

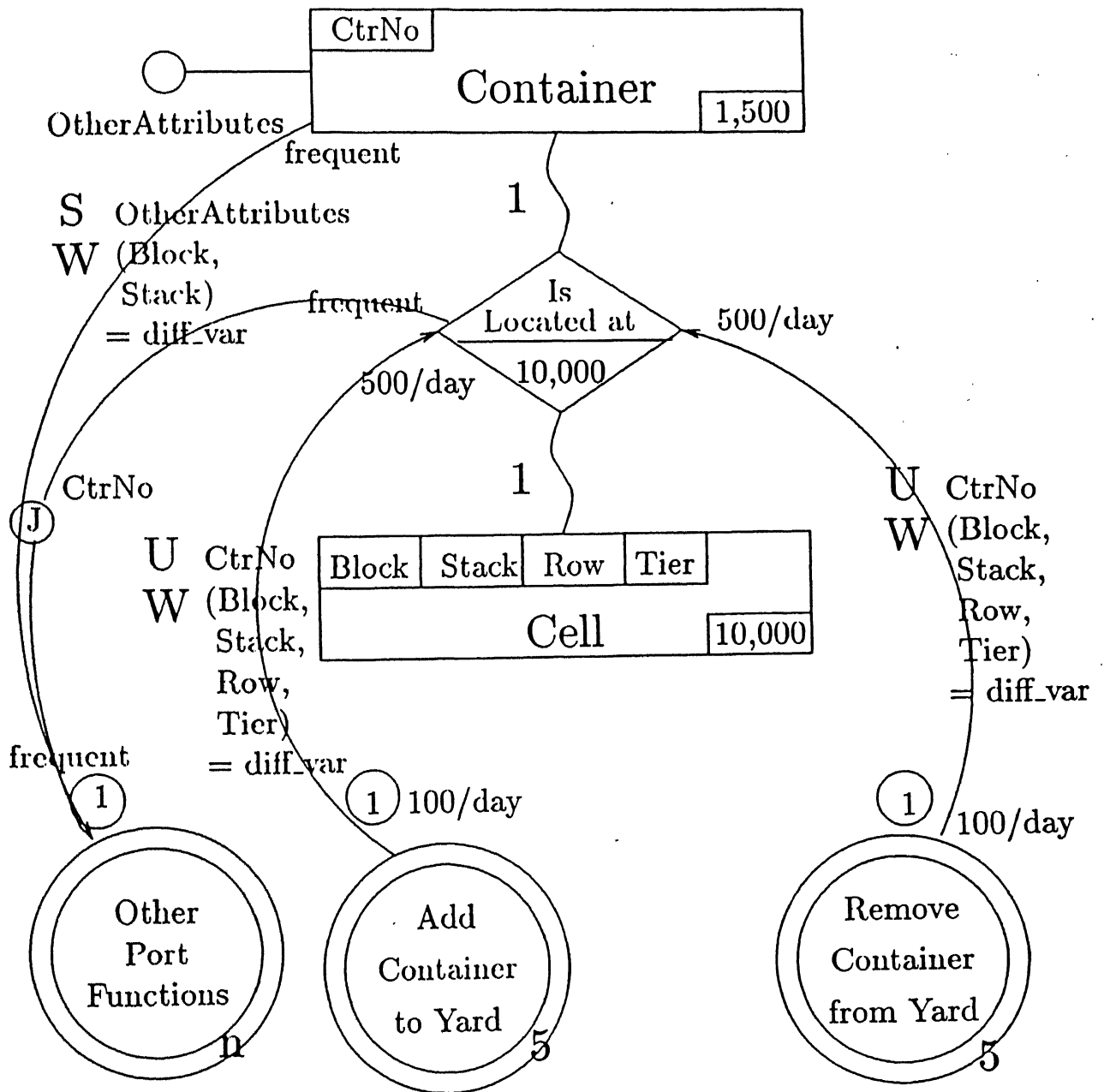


Figure 7.11: An "Update-only" Approach to Relation Usage

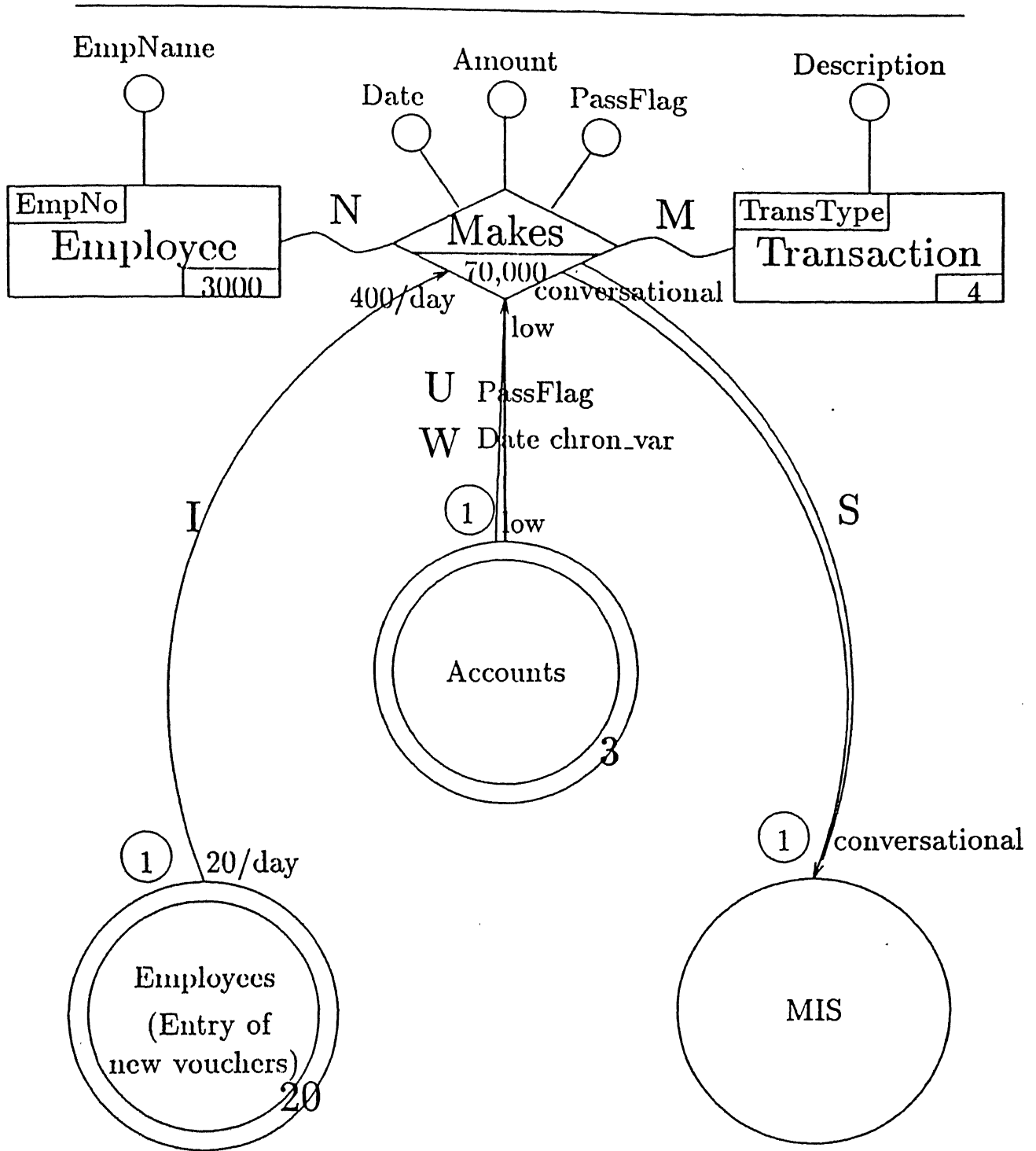


Figure 7.12: Operations versus MIS

Chapter 8

Limitations of the Diagramming Tool

We have found the benefits of the diagramming approach to be considerable. Many problem situations including potential deadlocks become *visually* obvious using this technique. This in itself is a boon, even if the tool does not recommend solutions, because the designer's attention gets drawn to those areas that need special treatment. This, in combination with the heuristics that we have listed, help a designer in choosing solutions that improve performance under concurrent access.

Of course, the tool is not perfect, and some of the limitations that we have found are the following:

- It is fairly good at modelling conflict between concurrent transactions *of the same type*, but is not so good at depicting conflicts between transactions of *different types*.
- At present, it does not cater to more complex queries, i.e. nested "where" clauses.
- Only *schema* design decisions can be *derived*. Transaction design and usage design decisions cannot be derived. Human innovation is still required in such cases, leading one to assume that it will be some time before a fully-automated database designer becomes a reality (In the meantime, we have classified the decisions that we know about into a set of rules. This rule set can be augmented as newer solutions are found).
- For any system with more than a few entities, relationships and transactions, the

diagram tends to become cluttered, defeating the original objective of visual, intuitive highlighting of potential performance problems.

Nevertheless, in spite of these limitations, we believe that this approach is a novel one that contributes in significant measure to the problem of *performance-oriented* database design.

Chapter 9

PODADE - A Computer-based Aid to Database Design

We have developed a set of programs which aid an application designer in making decisions based on performance considerations. We call it PODADE (Performance-Oriented Database application Design Engine).¹

9.1 Architecture and data structures

We found to our pleasant surprise, that the entire ERTD concept could be modelled very naturally with an ERD (see Fig. 9.1). The data structures then follow logically. Incidentally, since the design tool is most likely to be single-user, it was not necessary to model it using an ERTD!

The system has been implemented on top of the Ingres Database package and is entirely form- and menu-driven. Some procedures have been written in C with embedded SQL statements. The rest of the code is in the Ingres 4GL (OSL).

9.2 Brief logic

The tool runs in several phases. In the first phase, it captures information relating to the data side of an application alone, i.e. it notes the various entities and relationships, the

¹Any suggestion that this may be related to the Tamil phrase "Po da daiy!" (Get lost, you useless fellow!) is hereby strongly refuted

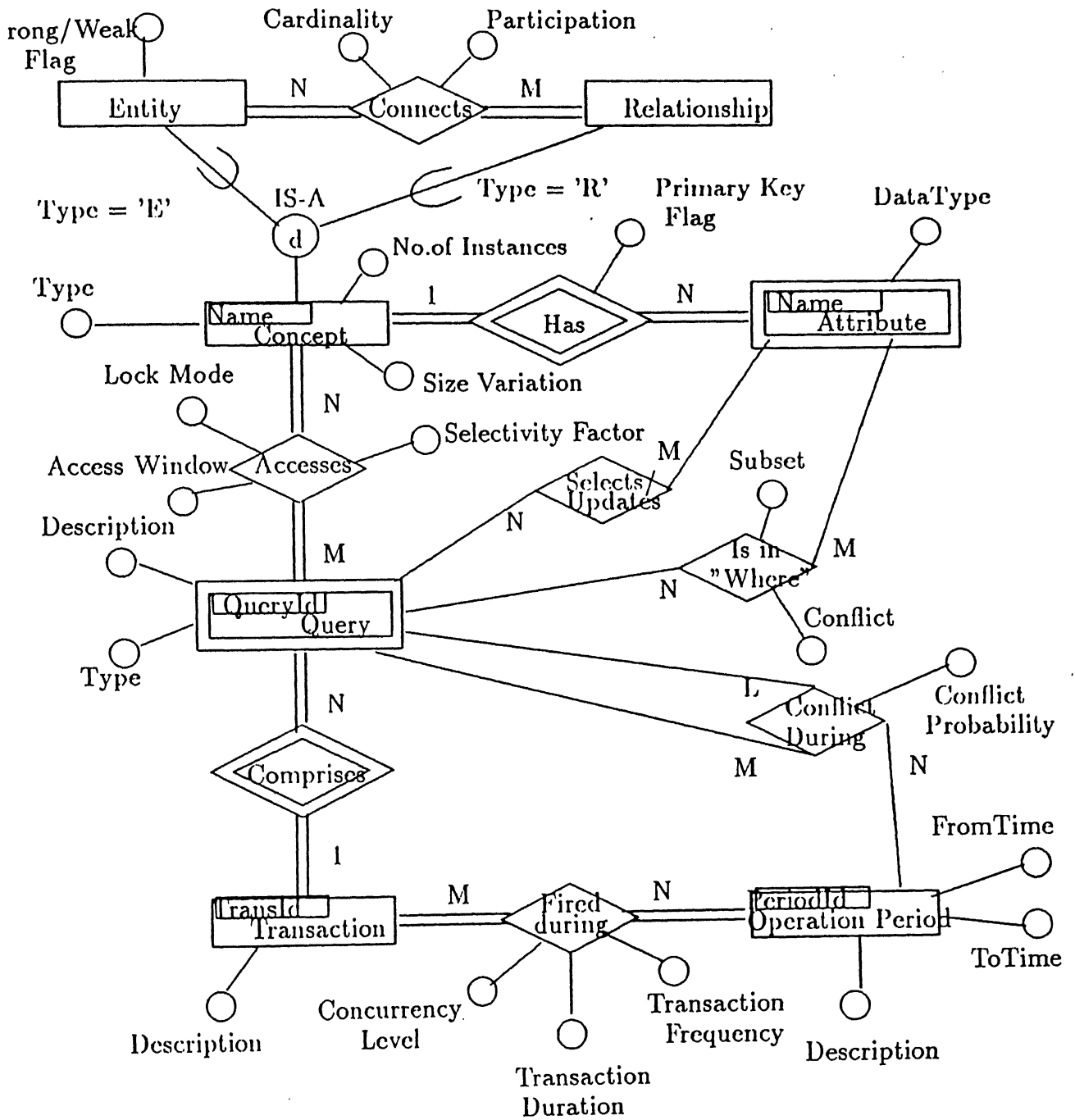


Figure 9.1: ERD of the Design Tool

way they are connected, their key attributes and other attributes. Weak entities, ternary and higher-order relationships, cardinality ratios and participation can all be modelled. The data types of attributes are also intelligently assumed provided the designer ends their names with some standard suffixes like `_NAME` (25 characters) and `_QTY` (4 byte integer).

In the second phase, it generates a design in the form of relations (in Third Normal Form) and relation attributes, based on the ERD captured. It even merges relations where possible.

In the third phase, it captures size and access information. How many instances of each entity/relationship are there likely to be and how quickly are they likely to grow/shrink? What are the various periods of the day when distinct access patterns can be seen? In each of these periods, what are the major transactions that are run? How frequently is each transaction run, by how many concurrent users, and how long is each transaction? The answers to all these are captured in a qualitative rather than in a quantitative form. PODADE also requests information on the queries that make up a transaction. What entities and relationships does a query access, in what lock mode? Which are the attributes accessed? Which are the attributes specified in the "where" clause of a query? What subset of an entity/relationship is accessed? Is this subset likely to overlap that accessed by another query?

During this phase, the tool builds up an exhaustive picture of how the database is likely to be accessed.

In the fourth and final phase, the tool provides many insights into the application for the benefit of the designer. For instance, it shows which are the largest tables, ordered by rate of growth. The designer is put on guard about potentially space-guzzling relations. It also shows the bottleneck relations in each operation period. It suggests techniques based on the heuristics described earlier to ameliorate such problems.

Chapter 10

Summary and Future Work

In this thesis we have described a problem, i.e. that of generating a design for a database application that would demonstrate adequate performance in a production environment. We had surveyed some existing tools and techniques and contrasted the approach of such methods with that employed by designers and implementors on some real-life projects. We found that most performance problems were being caused by lock contention between concurrent processes and transactions. The techniques used by human designers were aimed at solving such problems, but surprisingly, none of the formal techniques approached the problem from this angle, with the result that designs resulting from such techniques were often demonstrably inadequate from a performance viewpoint. We had expressed the need for a design tool that would *systematically* do what human designers were intuitively doing.

We then presented our attempt at developing such a technique.

The major contributions of this thesis are as follows :

It identifies the real problem of performance as being lock contention between concurrent transactions. Our diagrammatic technique is intuitive and models data and transactions together in a natural way. It makes conflict areas stand out visually. Even potential deadlocks can be detected in many cases by an examination of the diagram. A set of heuristics (rules of thumb) used by experienced designers to ease bottlenecks and improve concurrent performance has also been compiled from real-life experience. The technique has been tested on a computer-based tool and found to work satisfactorily.

Much remains to be done. Most importantly, the heuristic rules need to be made more formal and comprehensive, the notation for lock conflict needs to be improved to include

conflicts between transactions of different types, a way needs to be found to estimate system parameters like CPU power, main memory, disk space, etc. of the computer that would optimally run an application, etc.

One advantage that a diagramming technique has over a computer-based tool is that it does not require a designer to access a computer whenever a new idea is to be tried out. Especially in countries and organisations where computing resources are still scarce, a technique that requires no computing hardware would be highly prized. Hence our emphasis on the diagramming technique.

However, one of the major drawbacks of a pencil-and-paper approach, as we pointed out earlier, is that the diagram gets cluttered fairly quickly, and so the method fails to scale up well to larger systems, which form the target application for the technique. Validations and reworking of diagrams are also easier in a computer-based tool. With such a tool, perhaps even novice database designers would be able to produce designs comparable to those produced by more experienced professionals. Our aim has been to contribute towards such an improvement in quality.

Bibliography

- [ANTO85] De Antonellis V. and Di Leva A., "DATAID-1: A Database Design Methodology", *Information Systems*, Vol. 10, No. 2, 1985, pp. 181-195
together with
De Antonellis V. and Di Leva A., "A Case Study of Database Design using the DATAID Approach", *Information Systems*, Vol. 10, No. 3, 1985, pp. 339-359.
- [AGRA85] Agrawal R., Carey M.J. and Livny M., "Models for Studying Concurrency Control Performance: Alternatives and Implications", *Proc. Intl. Conf. on Management of Data*, 1985, pp. 108-121.
- [BATI92] Batini C., Ceri S. and Navathe S.B., "Conceptual Database Design - An Entity-Relationship Approach", The Benjamin/Cummins Publishing Company, Inc., 1992.
- [CHO 89] Cho H.R., Park S.J. and Hevia E., "An Approach to Efficient Database Design Incorporating Usage Information", *Information Systems*, Vol. 14, No. 2, 1989, pp. 107-115.
- [CMC 93] Interviews and private conversation with systems analysts, project leaders and database administrators of CMC Ltd., India, May - June, 1993.
- [ELLI79] Ellis C.A., "Information Control Nets: A Mathematical Model of Office Information Flow", *Proc. ACM Conf. on Simulation Modeling and Measurement of Computer Systems*, 1979.
- [ELMA89] Elmasri R. and Navathe S.B., "Fundamentals of Database Systems", Benjamin/Cummins Publishing Company, Inc., 1989, pp. 427-430.

- [FINK88] Finkelstein S., Schkolnick M. and Tiberio P., "Physical Database Design for Relational Databases", *ACM Trans. Database Syst.*, Vol. 13, No. 1, 1988, pp. 91-128.
- [GCP93a] G.C.Prasad and T.V.Prabhakar, "The Effect of Concurrency on Relational Database Design", *Technical Report TRCS-93-189*, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, September 1993.
- [GCP93b] G.C.Prasad and T.V.Prabhakar, "A Diagrammatic Aid to Performance-Oriented Database Design", *Technical Report TRCS-93-207*, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, November 1993.
- [GORL90] Gorla N. and Boe W., "Effect of Schema Size on Fragmentation Design in Multirelational Databases", *Information Systems*, Vol. 15, No. 3, 1990, pp. 291-301.
- [GRAY78] Gray J.N., "Notes on Database Operating Systems", *Lecture Notes on Computer Science*, Vol. 60, Springer, New York, pp. 394-481.
- [HABE90] Haberhauer F., "Physical Database Design Aspects of Relational DBMS Implementations", *Information Systems*, Vol. 15, No. 3, 1990, pp. 375-389.
- [HAMM79] Hammer M. and Niamir B., "A Heuristic Approach to Attribute Partitioning", *Proc. Intl. Conf. on Management of Data*, 1979, pp. 93-101.
- [JOHN81] Johnson R., "Modelling Summary Data", *Proc. Intl. Conf. on Management of Data*, 1981, pp.93-97.
- [KORT91] Korth H.F. and Silbershatz R., "Database System Concepts", 2nd Edition, McGraw-Hill, 1991.
- [LEON81] Leonard M. and Luong B.T., "Information Systems Design Approach Integrating Data and Transactions", *Proc. Intl. Conf. on Very Large Data Bases*, 1981, pp. 235-245.

- [NAVA89] Navathe S.B. and Ra M., "Vertical Partitioning for Database Design: A Graphical Algorithm", *Proc. Intl. Conf. on Management of Data*, 1989, pp. 440-450.
- [OZSO85] Ozsoyoglu G., Ozsoyoglu Z.M. and Mata F., "A Language and a Physical Organisation Technique for Summary Tables", *Proc. Intl. Conf. on Management of Data*, 1985, pp. 3-16.
- [PEIN88] Peinl P., Reuter A. and Sammer H., "High Contention in a Stock Trading Database: A Case Study", *Proc. Intl. Conf. on Management of Data*, 1988, pp. 260-268.
- [PETE77] Peterson J.R. "Petri Nets", *Computing Surveys* 9, No. 3, 1977, pp. 223-52.
- [RAM 89] Ram S. and Curran S.M., "An Automated Tool for Relational Database Design", *Information Systems*, Vol. 14, No. 3, 1989, pp. 247-259.
- [ROZE91] Rozen S. and Shasha D., "A Framework for Automating Physical Database Design", *Proc. Intl. Conf. on Very Large Data Bases*, 1991, pp. 401-411.
- [SAKA81] Sakai H., "A Method for Defining Information Structures and Transactions in Conceptual Schema Design", *Proc. Intl. Conf. on Very Large Data Bases*, 1981, pp.225-234.
- [SHAS92] Shasha D., Simon E. and Valduriez P., "Simple Rational Guidance for Chopping up Transactions", *Proc. Intl. Conf. on Management of Data*, 1992.
- [SRIN91] Srinivasan V. and Carey M.J., "Performance of B-Tree Concurrency Control Algorithms", *Proc. Intl. Conf. on Management of Data*, 1991, pp. 416-425.
- [WHAN81] Whang K-Y. and Sagalowicz D., "Separability - An Approach to Physical Database Design", *Proc. Intl. Conf. on Very Large Data Bases*, 1981, pp. 320-332.

Appendix A

Summary of Problems and Solutions (Schema Design)

<i>SITUATION</i>	<i>PROBLEM</i>	<i>SOLUTION</i>
Insertion of records by many users into the same table (Eg: Stock exchange, ledger postings)	Lock contention for last page.	Horizontal partitioning of table based on attribute value.
Groups of attributes in a relation accessed by different transactions (all of high frequency).	Contention for records and pages	Vertical partitioning of table with each set of attributes in a different relation.
Transactions inserting or updating records while others are making aggregation queries on the table. (Eg: Stock exchange, Serial no. generation)	Aggregation takes a long time, transaction duration is high. Conflicts between readers and writers.	Storage of aggregate values redundantly.

<i>SITUATION</i>	<i>PROBLEM</i>	<i>SOLUTION</i>
Different users access logically disjoint subsets of a table. (Eg: Department-based queries).	In page-locking systems, conflicts are possible. Group selects are slow.	Clustering records based on attribute that logically separates them.
Concurrent users access single records found close together (Eg: Rail reservation confirmation queries).	Access conflicts in page-locking systems.	Hashing/de-clustering of records on key value.
Frequent updates to a table with many indexes.	Index update causes contention for locks on index.	Use of fewer indexes.

Appendix B

Summary of Problems and Solutions (Transaction Design)

<i>SITUATION</i>	<i>PROBLEM</i>	<i>SOLUTION</i>
Transactions read value and conditionally update it. (Eg: Stock exchange)	Reader-writer contention.	Reading without locks. When needed, redo with locks.
Long transactions where absolute serialisability is not required.	High contention for locks.	Transaction to be split into smaller ones.
Records often selected in groups on one attribute, but others change frequently (Eg: Port Yard inventory).	"Scattering" of records with high access time per group select. Lock contention.	Pre-allocation of records and clustering by grouping attribute. Other attributes to be updated.

<i>SITUATION</i>	<i>PROBLEM</i>	<i>SOLUTION</i>
Writer transactions need latest data only. Readers need full data. (Eg: Operations and MIS queries)	Reads are slow. Readers block writers. Writers have to search for their records.	Older records to be archived. View provided to readers on union of old and new tables.